Command line API Reference

Table of contents

1. Introduction

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

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

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

2. Using the Yocto-Demo in command line

When you want to perform a punctual operation on your Yocto-Demo, such as reading a value, assigning a logical name, and so on, you can obviously use the Virtual Hub, but there is a simpler, faster, and more efficient method: the command line API.

The command line API is a set of executables, one by type of functionality offered by the range of Yoctopuce products. These executables are provided pre-compiled for all the Yoctopuce officially supported platforms/OS. Naturally, the executable sources are also provided[1].

2.1. Installing

Download the command line API[2]. You do not need to run any setup, simply copy the executables corresponding to your platform/OS in a directory of your choice. You may add this directory to your PATH variable to be able to access these executables from anywhere. You are all set, you only need to connect your Yocto-Demo, open a shell, and start working by typing for example:

```
C:\>YLed any set_power ON

C:\>YLed any set_blinking RELAX
```

To use the command API on Linux, you need either have root privileges or to define an *udev* rule for your system. See the *Troubleshooting* chapter for more details.

2.2. Use: general description

All the command line API executables work on the same principle. They must be called the following way

```
C:\>Executable [options] [target] command [parameter]
```

`[options]` manage the global workings of the commands, they allow you, for instance, to pilot a module remotely through the network, or to force the module to save its configuration after executing the command.

`[target]` is the name of the module or of the function to which the command applies. Some very generic commands do not need a target. You can also use the aliases "*any*" and "*all*", or a list of names separated by comas without space.

[1] If you want to recompile the command line API, you also need the C++ API.
[2] http://www.yoctopuce.com/EN/libraries.php

`command` is the command you want to run. Almost all the functions available in the classic programming APIs are available as commands. You need to respect neither the case nor the underlined characters in the command name.

`[parameters]` logically are the parameters needed by the command.

At any time, the command line API executables can provide a rather detailed help. Use for instance:

```
C:\>executable /help
```

to know the list of available commands for a given command line API executable, or even:

```
C:\>executable command /help
```

to obtain a detailed description of the parameters of a command.

2.3. Control of the Led function

To control the Led function of your Yocto-Demo, you need the YLed executable file.

For instance, you can launch:

```
C:\>YLed any set_power ON

C:\>YLed any set_blinking RELAX
```

This example uses the "*any*" target to indicate that we want to work on the first Led function found among all those available on the connected Yoctopuce modules when running. This prevents you from having to know the exact names of your function and of your module.

But you can use logical names as well, as long as you have configured them beforehand. Let us imagine a Yocto-Demo module with the *YCTOPOC1-123456* serial number which you have called "*MyModule*", and its led function which you have renamed "*MyFunction*". The five following calls are strictly equivalent (as long as *MyFunction* is defined only once, to avoid any ambiguity).

```
C:\>YLed YCTOPOC1-123456.led describe

C:\>YLed YCTOPOC1-123456.MyFunction describe

C:\>YLed MyModule.led describe

C:\>YLed MyModule.MyFunction describe

C:\>YLed MyFunction describe
```

To work on all the Led functions at the same time, use the "*all*" target.

```
C:\>YLed all describe
```

For more details on the possibilities of the `YLed` executable, use:

```
C:\>YLed /help
```

2.4. Control of the module part

Each module can be controlled in a similar way with the help of the `YModule` executable. For example, to obtain the list of all the connected modules, use:

```
C:\>YModule inventory
```

You can also use the following command to obtain an even more detailed list of the connected modules:

```
C:\>YModule all describe
```

Each `xxx` property of the module can be obtained thanks to a command of the `get_xxxx()` type, and the properties which are not read only can be modified with the `set_xxx()` command. For example:

```
C:\>YModule YCTOPOC1-12346 set_logicalName MonPremierModule

C:\>YModule YCTOPOC1-12346 get_logicalName
```

Changing the settings of the module

When you want to change the settings of a module, simply use the corresponding `set_xxx` command. However, this change happens only in the module RAM: if the module restarts, the changes are lost. To store them permanently, you must tell the module to save its current configuration in its nonvolatile memory. To do so, use the `saveToFlash` command. Inversely, it is possible to force the module to forget its current settings by using the `revertFromFlash` method. For example:

```
C:\>YModule YCTOPOC1-12346 set_logicalName MonPremierModule
C:\>YModule YCTOPOC1-12346 saveToFlash
```

Note that you can do the same thing in a single command with the `-s` option.

```
C:\>YModule -s  YCTOPOC1-12346 set_logicalName MonPremierModule
```

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.

2.5. Limitations

The command line API has the same limitation than the other APIs: there can be only one application at a given time which can access the modules natively. By default, the command line API works in native mode.

You can easily work around this limitation by using a Virtual Hub: run the VirtualHub[3] on the concerned machine, and use the executables of the command line API with the `-r` option. For example, if you use:

```
C:\>YModule  inventory
```

you obtain a list of the modules connected by USB, using a native access. If another command which accesses the modules natively is already running, this does not work. But if you run a Virtual Hub, and you give your command in the form:

```
C:\>YModule -r 127.0.0.1 inventory
```

it works because the command is not executed natively anymore, but through the Virtual Hub. Note that the Virtual Hub counts as a native application.

[3] http://www.yoctopuce.com/EN/virtualhub.php

3. Reference

3.1. General functions

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

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

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

Global functions
yCheckLogicalName(**name**)
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**)

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

yRegisterLogFunction(**logfun**)

Registers a log callback function.

ySelectArchitecture(**arch**)

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

ySetDelegate(**object**)

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

ySetTimeout(**callback**, **ms_timeout**, **arguments**)

Invoke the specified callback function after a given timeout.

ySleep(**ms_duration**, **errmsg**)

Pauses the execution flow for a specified duration.

yTriggerHubDiscovery(**errmsg**)

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

yUnregisterHub(**url**)

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

yUpdateDeviceList(**errmsg**)

Triggers a (re)detection of connected Yoctopuce modules.

yUpdateDeviceList_async(**callback**, **context**)

Triggers a (re)detection of connected Yoctopuce modules.

3.2. Accelerometer function interface

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

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

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

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

accelerometer→get_highestValue()

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

accelerometer→get_logFrequency()

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

accelerometer→get_logicalName()

Returns the logical name of the accelerometer.

accelerometer→get_lowestValue()

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

accelerometer→get_module()

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

accelerometer→get_module_async(callback, context)

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

accelerometer→get_recordedData(startTime, endTime)

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

accelerometer→get_reportFrequency()

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

accelerometer→get_resolution()

Returns the resolution of the measured values.

accelerometer→get_unit()

Returns the measuring unit for the acceleration.

accelerometer→get_userData()

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

accelerometer→get_xValue()

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

accelerometer→get_yValue()

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

accelerometer→get_zValue()

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

accelerometer→isOnline()

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

accelerometer→isOnline_async(callback, context)

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

accelerometer→load(msValidity)

Preloads the accelerometer cache with a specified validity duration.

accelerometer→loadCalibrationPoints(rawValues, refValues)

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

accelerometer→load_async(msValidity, callback, context)

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

accelerometer→nextAccelerometer()

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

accelerometer→registerTimedReportCallback(callback)

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

accelerometer→registerValueCallback(callback)

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

accelerometer→set_highestValue(newval)

Changes the recorded maximal value observed.

accelerometer→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

accelerometer→set_logicalName(newval)

Changes the logical name of the accelerometer.

accelerometer→set_lowestValue(newval)

Changes the recorded minimal value observed.

accelerometer→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

accelerometer→set_resolution(newval)

Changes the resolution of the measured physical values.

accelerometer→set_userData(data)

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

accelerometer→wait_async(callback, context)

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

accelerometer→calibrateFromPoints() YAccelerometer
YAccelerometer calibrateFromPoints

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

YAccelerometer **target calibrateFromPoints rawValues refValues**

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

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

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

accelerometer→get_advertisedValue() YAccelerometer
accelerometer→advertisedValue()YAccelerometer
get_advertisedValue

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

YAccelerometer **target** **get_advertisedValue**

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

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

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

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

YAccelerometer **target** **get_currentRawValue**

Returns :

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

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

accelerometer→get_currentValue() YAccelerometer
accelerometer→currentValue()YAccelerometer
get_currentValue

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

YAccelerometer **target** **get_currentValue**

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

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

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

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

YAccelerometer **target get_highestValue**

Returns :

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

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

accelerometer→get_logFrequency() YAccelerometer
accelerometer→logFrequency()YAccelerometer
get_logFrequency

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

YAccelerometer **target** **get_logFrequency**

Returns :

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

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

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

Returns the logical name of the accelerometer.

YAccelerometer **target get_logicalName**

Returns :
a string corresponding to the logical name of the accelerometer.

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

accelerometer→get_lowestValue() YAccelerometer
accelerometer→lowestValue()YAccelerometer
get_lowestValue

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

YAccelerometer **target** **get_lowestValue**

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

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

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

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

YAccelerometer **target get_recordedData startTime endTime**

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

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

Parameters :

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

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

Returns :

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

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

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

YAccelerometer **target** **get_reportFrequency**

Returns :

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

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

accelerometer→get_resolution() YAccelerometer
accelerometer→resolution()YAccelerometer
get_resolution

Returns the resolution of the measured values.

```
YAccelerometer target get_resolution
```

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

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

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

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

Returns the measuring unit for the acceleration.

```
YAccelerometer target get_unit
```

Returns :
a string corresponding to the measuring unit for the acceleration

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

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

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

YAccelerometer **target** **get_xValue**

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

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

accelerometer→get_yValue() YAccelerometer
accelerometer→yValue()YAccelerometer get_yValue

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

```
YAccelerometer target get_yValue
```

Returns :

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

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

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

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

```
YAccelerometer target get_zValue
```

Returns :

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

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

accelerometer→loadCalibrationPoints() YAccelerometer
YAccelerometer loadCalibrationPoints

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

YAccelerometer **target loadCalibrationPoints rawValues refValues**

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

accelerometer→set_highestValue() YAccelerometer
accelerometer→setHighestValue()YAccelerometer
set_highestValue

Changes the recorded maximal value observed.

YAccelerometer **target set_highestValue newval**

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the datalogger recording frequency for this function.

```
YAccelerometer target set_logFrequency newval
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

accelerometer→set_logicalName() YAccelerometer
accelerometer→setLogicalName()YAccelerometer
set_logicalName

Changes the logical name of the accelerometer.

```
YAccelerometer 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 accelerometer.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

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

Changes the recorded minimal value observed.

YAccelerometer **target set_lowestValue newval**

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the timed value notification frequency for this function.

```
YAccelerometer target set_reportFrequency newval
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

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

Changes the resolution of the measured physical values.

```
YAccelerometer target set_resolution newval
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

3.3. Altitude function interface

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

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

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

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

altitude→get_highestValue()

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

altitude→get_logFrequency()

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

altitude→get_logicalName()

Returns the logical name of the altimeter.

altitude→get_lowestValue()

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

altitude→get_module()

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

altitude→get_module_async(callback, context)

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

altitude→get_qnh()

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

altitude→get_recordedData(startTime, endTime)

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

altitude→get_reportFrequency()

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

altitude→get_resolution()

Returns the resolution of the measured values.

altitude→get_unit()

Returns the measuring unit for the altitude.

altitude→get_userData()

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

altitude→isOnline()

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

altitude→isOnline_async(callback, context)

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

altitude→load(msValidity)

Preloads the altimeter cache with a specified validity duration.

altitude→loadCalibrationPoints(rawValues, refValues)

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

altitude→load_async(msValidity, callback, context)

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

altitude→nextAltitude()

Continues the enumeration of altimeters started using `yFirstAltitude()`.

altitude→registerTimedReportCallback(callback)

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

altitude→registerValueCallback(callback)

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

altitude→set_currentValue(newval)

Changes the current estimated altitude.

altitude→set_highestValue(newval)

Changes the recorded maximal value observed.

altitude→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

altitude→set_logicalName(newval)

Changes the logical name of the altimeter.

altitude→set_lowestValue(newval)

Changes the recorded minimal value observed.

altitude→set_qnh(newval)

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

altitude→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

altitude→set_resolution(newval)

Changes the resolution of the measured physical values.

altitude→set_userData(data)

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

altitude→wait_async(callback, context)

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

altitude→calibrateFromPoints()YAltitude calibrateFromPoints YAltitude

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

YAltitude **target calibrateFromPoints rawValues refValues**

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

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

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

altitude→get_advertisedValue() YAltitude
altitude→advertisedValue()YAltitude
get_advertisedValue

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

YAltitude **target** **get_advertisedValue**

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

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

altitude→get_currentRawValue() YAltitude
altitude→currentRawValue()YAltitude
get_currentRawValue

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

YAltitude **target get_currentRawValue**

Returns :

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

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

altitude→get_currentValue() YAltitude
altitude→currentValue()YAltitude get_currentValue

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

YAltitude **target** **get_currentValue**

Returns :

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

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

altitude→get_highestValue() YAltitude
altitude→highestValue()YAltitude get_highestValue

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

YAltitude **target get_highestValue**

Returns :

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

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

altitude→get_logFrequency() YAltitude
altitude→logFrequency()YAltitude get_logFrequency

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

YAltitude **target** **get_logFrequency**

Returns :

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

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

altitude→get_logicalName() YAltitude
altitude→logicalName()YAltitude get_logicalName

Returns the logical name of the altimeter.

YAltitude **target get_logicalName**

Returns :
a string corresponding to the logical name of the altimeter.

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

altitude→get_lowestValue() YAltitude
altitude→lowestValue()YAltitude get_lowestValue

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

YAltitude **target** **get_lowestValue**

Returns :

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

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

altitude→get_qnh() YAltitude
altitude→qnh()YAltitude get_qnh

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

```
YAltitude target get_qnh
```

Returns :

a floating point number corresponding to the barometric pressure adjusted to sea level used to compute the altitude (QNH)

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

altitude→get_recordedData() YAltitude
altitude→recordedData()YAltitude get_recordedData

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

YAltitude **target get_recordedData startTime endTime**

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

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

Parameters :

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

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

Returns :

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

altitude→get_reportFrequency() YAltitude
altitude→reportFrequency()YAltitude
get_reportFrequency

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

YAltitude **target get_reportFrequency**

Returns :

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

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

altitude→get_resolution() YAltitude altitude→resolution()YAltitude get_resolution

Returns the resolution of the measured values.

```
YAltitude target get_resolution
```

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

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

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

altitude→get_unit() YAltitude
altitude→unit()YAltitude get_unit

Returns the measuring unit for the altitude.

```
YAltitude target get_unit
```

Returns :
a string corresponding to the measuring unit for the altitude

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

altitude→loadCalibrationPoints()YAltitude YAltitude loadCalibrationPoints

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

YAltitude **target** **loadCalibrationPoints** **rawValues** **refValues**

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

altitude→set_currentValue() YAltitude
altitude→setCurrentValue()YAltitude
set_currentValue

Changes the current estimated altitude.

```
YAltitude target set_currentValue newval
```

This allows to compensate for ambient pressure variations and to work in relative mode.

Parameters :

newval a floating point number corresponding to the current estimated altitude

Returns :

YAPI_SUCCESS if the call succeeds.

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

altitude→set_highestValue() YAltitude
altitude→setHighestValue()YAltitude
set_highestValue

Changes the recorded maximal value observed.

```
YAltitude target set_highestValue newval
```

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

altitude→set_logFrequency() YAltitude
altitude→setLogFrequency()YAltitude
set_logFrequency

Changes the datalogger recording frequency for this function.

```
YAltitude target set_logFrequency newval
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

altitude→set_logicalName() YAltitude
altitude→setLogicalName()YAltitude
set_logicalName

Changes the logical name of the altimeter.

```
YAltitude 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 altimeter.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

altitude→set_lowestValue() YAltitude
altitude→setLowestValue()YAltitude set_lowestValue

Changes the recorded minimal value observed.

```
YAltitude target set_lowestValue newval
```

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

altitude→set_qnh() YAltitude
altitude→setQnh()YAltitude set_qnh

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

```
YAltitude target set_qnh newval
```

This enables you to compensate for atmospheric pressure changes due to weather conditions.

Parameters :

newval a floating point number corresponding to the barometric pressure adjusted to sea level used to compute the altitude (QNH)

Returns :

YAPI_SUCCESS if the call succeeds.

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

altitude→set_reportFrequency() YAltitude
altitude→setReportFrequency()YAltitude
set_reportFrequency

Changes the timed value notification frequency for this function.

```
YAltitude target set_reportFrequency newval
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

altitude→set_resolution() YAltitude altitude→setResolution()YAltitude set_resolution

Changes the resolution of the measured physical values.

```
YAltitude target set_resolution newval
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

3.4. AnButton function interface

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

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

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

Global functions
yFindAnButton(**func**) Retrieves an analog input for a given identifier.
yFirstAnButton() Starts the enumeration of analog inputs currently accessible.
YAnButton methods
anbutton→describe() Returns a short text that describes unambiguously the instance of the analog input in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
anbutton→get_advertisedValue() Returns the current value of the analog input (no more than 6 characters).
anbutton→get_analogCalibration() Tells if a calibration process is currently ongoing.
anbutton→get_calibratedValue() Returns the current calibrated input value (between 0 and 1000, included).
anbutton→get_calibrationMax() Returns the maximal value measured during the calibration (between 0 and 4095, included).
anbutton→get_calibrationMin() Returns the minimal value measured during the calibration (between 0 and 4095, included).
anbutton→get_errorMessage() Returns the error message of the latest error with the analog input.
anbutton→get_errorType() Returns the numerical error code of the latest error with the analog input.
anbutton→get_friendlyName() Returns a global identifier of the analog input in the format `MODULE_NAME.FUNCTION_NAME`.
anbutton→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

anbutton→get_functionId()

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

anbutton→get_hardwareId()

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

anbutton→get_isPressed()

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

anbutton→get_lastTimePressed()

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

anbutton→get_lastTimeReleased()

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

anbutton→get_logicalName()

Returns the logical name of the analog input.

anbutton→get_module()

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

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

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

anbutton→get_pulseCounter()

Returns the pulse counter value

anbutton→get_pulseTimer()

Returns the timer of the pulses counter (ms)

anbutton→get_rawValue()

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

anbutton→get_sensitivity()

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

anbutton→get_userData()

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

anbutton→isOnline()

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

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

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

anbutton→load(**msValidity**)

Preloads the analog input cache with a specified validity duration.

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

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

anbutton→nextAnButton()

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

anbutton→registerValueCallback(**callback**)

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

anbutton→resetCounter()

Returns the pulse counter value as well as his timer

anbutton→set_analogCalibration(**newval**)

Starts or stops the calibration process.

anbutton→set_calibrationMax(**newval**)

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

anbutton→set_calibrationMin(newval)

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

anbutton→set_logicalName(newval)

Changes the logical name of the analog input.

anbutton→set_sensitivity(newval)

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

anbutton→set_userData(data)

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

anbutton→wait_async(callback, context)

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

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

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

YAnButton **target get_advertisedValue**

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

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

anbutton→get_analogCalibration() YAnButton
anbutton→analogCalibration()YAnButton
get_analogCalibration

Tells if a calibration process is currently ongoing.

YAnButton **target** **get_analogCalibration**

Returns :
either Y_ANALOGCALIBRATION_OFF or Y_ANALOGCALIBRATION_ON

On failure, throws an exception or returns Y_ANALOGCALIBRATION_INVALID.

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

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

YAnButton **target get_calibratedValue**

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

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

anbutton→get_calibrationMax() YAnButton
anbutton→calibrationMax()YAnButton
get_calibrationMax

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

YAnButton **target** **get_calibrationMax**

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

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

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

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

YAnButton **target get_calibrationMin**

Returns :

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

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

anbutton→get_isPressed() YAnButton
anbutton→isPressed()YAnButton get_isPressed

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

YAnButton **target** **get_isPressed**

Returns :

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

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

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

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

```
YAnButton target get_lastTimePressed
```

Returns :

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

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

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

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

YAnButton **target** **get_lastTimeReleased**

Returns :

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

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

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

Returns the logical name of the analog input.

YAnButton **target get_logicalName**

Returns :
a string corresponding to the logical name of the analog input.

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

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

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

YAnButton **target** **get_rawValue**

Returns :

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

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

anbutton→get_sensitivity() YAnButton
anbutton→sensitivity()YAnButton get_sensitivity

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

YAnButton **target get_sensitivity**

Returns :

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

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

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

Starts or stops the calibration process.

YAnButton **target** **set_analogCalibration** **newval**

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

Parameters :

newval either `Y_ANALOGCALIBRATION_OFF` or `Y_ANALOGCALIBRATION_ON`

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

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

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

```
YAnButton target set_calibrationMax newval
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

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

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

```
YAnButton target set_calibrationMin newval
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

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

Changes the logical name of the analog input.

```
YAnButton 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 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()YAnButton set_sensitivity

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

YAnButton **target set_sensitivity newval**

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

3.5. CarbonDioxide function interface

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

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

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

Global functions
yFindCarbonDioxide(**func**) Retrieves a CO2 sensor for a given identifier.
yFirstCarbonDioxide() Starts the enumeration of CO2 sensors currently accessible.
`YCarbonDioxide` methods
carbondioxide→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
carbondioxide→describe() Returns a short text that describes unambiguously the instance of the CO2 sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
carbondioxide→get_advertisedValue() Returns the current value of the CO2 sensor (no more than 6 characters).
carbondioxide→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in ppm (vol), as a floating point number.
carbondioxide→get_currentValue() Returns the current value of the CO2 concentration, in ppm (vol), as a floating point number.
carbondioxide→get_errorMessage() Returns the error message of the latest error with the CO2 sensor.
carbondioxide→get_errorType() Returns the numerical error code of the latest error with the CO2 sensor.
carbondioxide→get_friendlyName() Returns a global identifier of the CO2 sensor in the format `MODULE_NAME.FUNCTION_NAME`.
carbondioxide→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
carbondioxide→get_functionId() Returns the hardware identifier of the CO2 sensor, without reference to the module.
carbondioxide→get_hardwareId()

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

carbondioxide→get_highestValue()

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

carbondioxide→get_logFrequency()

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

carbondioxide→get_logicalName()

Returns the logical name of the CO2 sensor.

carbondioxide→get_lowestValue()

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

carbondioxide→get_module()

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

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

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

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

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

carbondioxide→get_reportFrequency()

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

carbondioxide→get_resolution()

Returns the resolution of the measured values.

carbondioxide→get_unit()

Returns the measuring unit for the CO2 concentration.

carbondioxide→get_userData()

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

carbondioxide→isOnline()

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

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

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

carbondioxide→load(**msValidity**)

Preloads the CO2 sensor cache with a specified validity duration.

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

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

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

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

carbondioxide→nextCarbonDioxide()

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

carbondioxide→registerTimedReportCallback(**callback**)

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

carbondioxide→registerValueCallback(**callback**)

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

carbondioxide→set_highestValue(**newval**)

Changes the recorded maximal value observed.

carbondioxide→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

carbondioxide→set_logicalName(**newval**)

Changes the logical name of the CO2 sensor.

carbondioxide→set_lowestValue(newval)

Changes the recorded minimal value observed.

carbondioxide→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

carbondioxide→set_resolution(newval)

Changes the resolution of the measured physical values.

carbondioxide→set_userData(data)

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

carbondioxide→wait_async(callback, context)

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

carbondioxide→calibrateFromPoints() YCarbonDioxide
YCarbonDioxide calibrateFromPoints

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

YCarbonDioxide **target calibrateFromPoints rawValues refValues**

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

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

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

carbondioxide→get_advertisedValue() YCarbonDioxide
carbondioxide→advertisedValue()YCarbonDioxide
get_advertisedValue

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

YCarbonDioxide **target get_advertisedValue**

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

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

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

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

YCarbonDioxide **target get_currentRawValue**

Returns :

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

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

carbondioxide→get_currentValue() YCarbonDioxide
carbondioxide→currentValue()YCarbonDioxide
get_currentValue

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

YCarbonDioxide **target** **get_currentValue**

Returns :

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

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

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

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

YCarbonDioxide **target get_highestValue**

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

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

carbondioxide→get_logFrequency() YCarbonDioxide
carbondioxide→logFrequency()YCarbonDioxide
get_logFrequency

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

YCarbonDioxide **target** **get_logFrequency**

Returns :

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

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

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

Returns the logical name of the CO2 sensor.

YCarbonDioxide **target get_logicalName**

Returns :
a string corresponding to the logical name of the CO2 sensor.

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

carbondioxide→get_lowestValue() YCarbonDioxide
carbondioxide→lowestValue()YCarbonDioxide
get_lowestValue

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

YCarbonDioxide **target** **get_lowestValue**

Returns :

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

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

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

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

YCarbonDioxide **target get_recordedData startTime endTime**

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

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

Parameters :

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

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

Returns :

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

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

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

YCarbonDioxide **target** **get_reportFrequency**

Returns :

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

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

carbondioxide→get_resolution() YCarbonDioxide
carbondioxide→resolution()YCarbonDioxide
get_resolution

Returns the resolution of the measured values.

YCarbonDioxide **target get_resolution**

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

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

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

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

Returns the measuring unit for the CO2 concentration.

```
YCarbonDioxide target get_unit
```

Returns :

a string corresponding to the measuring unit for the CO2 concentration

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

carbondioxide→loadCalibrationPoints() YCarbonDioxide
YCarbonDioxide loadCalibrationPoints

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

YCarbonDioxide **target loadCalibrationPoints rawValues refValues**

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

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

Changes the recorded maximal value observed.

```
YCarbonDioxide target set_highestValue newval
```

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_logFrequency() YCarbonDioxide
carbondioxide→setLogFrequency()YCarbonDioxide
set_logFrequency

Changes the datalogger recording frequency for this function.

```
YCarbonDioxide target set_logFrequency newval
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

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

Changes the logical name of the CO2 sensor.

```
YCarbonDioxide 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 CO2 sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

carbondioxide→set_lowestValue() YCarbonDioxide
carbondioxide→setLowestValue()YCarbonDioxide
set_lowestValue

Changes the recorded minimal value observed.

YCarbonDioxide **target set_lowestValue newval**

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the timed value notification frequency for this function.

```
YCarbonDioxide target set_reportFrequency newval
```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_resolution() YCarbonDioxide
carbondioxide→setResolution()YCarbonDioxide
set_resolution

Changes the resolution of the measured physical values.

```
YCarbonDioxide target set_resolution newval
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

3.6. ColorLed function interface

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

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

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

Global functions
yFindColorLed(**func**) Retrieves an RGB led for a given identifier.
yFirstColorLed() Starts the enumeration of RGB leds currently accessible.
`YColorLed` methods
colorled→describe() Returns a short text that describes unambiguously the instance of the RGB led in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
colorled→get_advertisedValue() Returns the current value of the RGB led (no more than 6 characters).
colorled→get_errorMessage() Returns the error message of the latest error with the RGB led.
colorled→get_errorType() Returns the numerical error code of the latest error with the RGB led.
colorled→get_friendlyName() Returns a global identifier of the RGB led in the format `MODULE_NAME.FUNCTION_NAME`.
colorled→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
colorled→get_functionId() Returns the hardware identifier of the RGB led, without reference to the module.
colorled→get_hardwareId() Returns the unique hardware identifier of the RGB led in the form `SERIAL.FUNCTIONID`.
colorled→get_hslColor() Returns the current HSL color of the led.
colorled→get_logicalName() Returns the logical name of the RGB led.

colorled→get_module()

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

colorled→get_module_async(callback, context)

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

colorled→get_rgbColor()

Returns the current RGB color of the led.

colorled→get_rgbColorAtPowerOn()

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

colorled→get_userData()

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

colorled→hslMove(hsl_target, ms_duration)

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

colorled→isOnline()

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

colorled→isOnline_async(callback, context)

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

colorled→load(msValidity)

Preloads the RGB led cache with a specified validity duration.

colorled→load_async(msValidity, callback, context)

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

colorled→nextColorLed()

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

colorled→registerValueCallback(callback)

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

colorled→rgbMove(rgb_target, ms_duration)

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

colorled→set_hslColor(newval)

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

colorled→set_logicalName(newval)

Changes the logical name of the RGB led.

colorled→set_rgbColor(newval)

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

colorled→set_rgbColorAtPowerOn(newval)

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

colorled→set_userData(data)

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

colorled→wait_async(callback, context)

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

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

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

YColorLed **target get_advertisedValue**

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

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

colorled→get_hslColor() YColorLed
colorled→hslColor()YColorLed get_hslColor

Returns the current HSL color of the led.

YColorLed **target** **get_hslColor**

Returns :
an integer corresponding to the current HSL color of the led

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

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

Returns the logical name of the RGB led.

YColorLed **target get_logicalName**

Returns :
a string corresponding to the logical name of the RGB led.

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

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

Returns the current RGB color of the led.

```
YColorLed target get_rgbColor
```

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

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

colorled→get_rgbColorAtPowerOn() YColorLed
colorled→rgbColorAtPowerOn()YColorLed
get_rgbColorAtPowerOn

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

```
YColorLed target get_rgbColorAtPowerOn
```

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

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

colorled→hslMove()YColorLed hslMove YColorLed

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

YColorLed **target hslMove hsl_target ms_duration**

Parameters :

hsl_target desired HSL color at the end of the transition

ms_duration duration of the transition, in millisecond

Returns :

YAPI_SUCCESS if the call succeeds.

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

colorled→rgbMove()YColorLed rgbMove YColorLed

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

YColorLed **target rgbMove rgb_target ms_duration**

Parameters :

rgb_target desired RGB color at the end of the transition

ms_duration duration of the transition, in millisecond

Returns :

YAPI_SUCCESS if the call succeeds.

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

colorled→set_hslColor() YColorLed
colorled→setHslColor()YColorLed set_hslColor

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

```
YColorLed target set_hslColor newval
```

Encoding is done as follows: 0xHHSSLL.

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the logical name of the RGB led.

```
YColorLed 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 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()YColorLed set_rgbColor

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

```
YColorLed target set_rgbColor newval
```

Encoding is done as follows: 0xRRGGBB.

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

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

```
YColorLed target set_rgbColorAtPowerOn newval
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

3.7. Compass function interface

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

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

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

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

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.

compass→calibrateFromPoints()YCompass YCompass calibrateFromPoints

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

YCompass **target calibrateFromPoints rawValues refValues**

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

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

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

compass→get_advertisedValue() YCompass
compass→advertisedValue()YCompass
get_advertisedValue

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

```
YCompass target get_advertisedValue
```

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

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

compass→get_currentRawValue() YCompass
compass→currentRawValue()YCompass
get_currentRawValue

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

YCompass **target** **get_currentRawValue**

Returns :

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

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

compass→get_currentValue() YCompass
compass→currentValue()YCompass
get_currentValue

Returns the current value of the relative bearing, in degrees, as a floating point number.

YCompass **target** **get_currentValue**

Returns :
a floating point number corresponding to the current value of the relative bearing, in degrees, as a floating point number

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

compass→get_highestValue() YCompass
compass→highestValue()YCompass
get_highestValue

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

YCompass **target** **get_highestValue**

Returns :

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

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

compass→get_logFrequency() YCompass
compass→logFrequency()YCompass
get_logFrequency

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

YCompass **target** **get_logFrequency**

Returns :

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

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

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

Returns the logical name of the compass.

YCompass **target get_logicalName**

Returns :
a string corresponding to the logical name of the compass.

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

compass→get_lowestValue() YCompass
compass→lowestValue()YCompass get_lowestValue

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

YCompass **target** **get_lowestValue**

Returns :

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

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

compass→get_magneticHeading() YCompass
compass→magneticHeading()YCompass
get_magneticHeading

Returns the magnetic heading, regardless of the configured bearing.

YCompass **target** **get_magneticHeading**

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

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

compass→get_recordedData() YCompass
compass→recordedData()YCompass
get_recordedData

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

YCompass **target get_recordedData startTime endTime**

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

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

Parameters :

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

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

Returns :

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

compass→get_reportFrequency() YCompass
compass→reportFrequency()YCompass
get_reportFrequency

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

YCompass **target** **get_reportFrequency**

Returns :

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

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

compass→get_resolution() YCompass compass→resolution()YCompass get_resolution

Returns the resolution of the measured values.

```
YCompass target get_resolution
```

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

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

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

compass→get_unit() YCompass
compass→unit()YCompass get_unit

Returns the measuring unit for the relative bearing.

YCompass **target** **get_unit**

Returns :
a string corresponding to the measuring unit for the relative bearing

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

compass→loadCalibrationPoints()YCompass YCompass loadCalibrationPoints

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

YCompass **target** **loadCalibrationPoints** **rawValues** **refValues**

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

compass→set_highestValue() YCompass
compass→setHighestValue()YCompass
set_highestValue

Changes the recorded maximal value observed.

```
YCompass target set_highestValue newval
```

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

compass→set_logFrequency() YCompass
compass→setLogFrequency()YCompass
set_logFrequency

Changes the datalogger recording frequency for this function.

```
YCompass target set_logFrequency newval
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

compass→set_logicalName() YCompass
compass→setLogicalName()YCompass
set_logicalName

Changes the logical name of the compass.

```
YCompass 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 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()YCompass
set_lowestValue

Changes the recorded minimal value observed.

```
YCompass target set_lowestValue newval
```

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

compass→set_reportFrequency() YCompass
compass→setReportFrequency()YCompass
set_reportFrequency

Changes the timed value notification frequency for this function.

```
YCompass target set_reportFrequency newval
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

compass→set_resolution() YCompass compass→setResolution()YCompass set_resolution

Changes the resolution of the measured physical values.

```
YCompass target set_resolution newval
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

3.8. Current function interface

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

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

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

Global functions

yFindCurrent(**func**)

Retrieves a current sensor for a given identifier.

yFirstCurrent()

Starts the enumeration of current sensors currently accessible.

`YCurrent` methods

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

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

current→describe()

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

current→get_advertisedValue()

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

current→get_currentRawValue()

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

current→get_currentValue()

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

current→get_errorMessage()

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

current→get_errorType()

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

current→get_friendlyName()

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

current→get_functionDescriptor()

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

current→get_functionId()

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

current→get_hardwareId()

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

current→get_highestValue()

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

current→get_logFrequency()

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

current→get_logicalName()

Returns the logical name of the current sensor.

current→get_lowestValue()

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

current→get_module()

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

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

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

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

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

current→get_reportFrequency()

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

current→get_resolution()

Returns the resolution of the measured values.

current→get_unit()

Returns the measuring unit for the current.

current→get_userData()

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

current→isOnline()

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

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

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

current→load(**msValidity**)

Preloads the current sensor cache with a specified validity duration.

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

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

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

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

current→nextCurrent()

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

current→registerTimedReportCallback(**callback**)

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

current→registerValueCallback(**callback**)

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

current→set_highestValue(**newval**)

Changes the recorded maximal value observed.

current→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

current→set_logicalName(**newval**)

Changes the logical name of the current sensor.

current→set_lowestValue(newval)

Changes the recorded minimal value observed.

current→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

current→set_resolution(newval)

Changes the resolution of the measured physical values.

current→set_userData(data)

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

current→wait_async(callback, context)

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

current→calibrateFromPoints()YCurrent calibrateFromPoints YCurrent

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

YCurrent **target calibrateFromPoints rawValues refValues**

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

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

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

current→get_advertisedValue() YCurrent
current→advertisedValue()YCurrent
get_advertisedValue

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

YCurrent **target** **get_advertisedValue**

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

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

current→get_currentRawValue() YCurrent
current→currentRawValue()YCurrent
get_currentRawValue

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

YCurrent **target get_currentRawValue**

Returns :

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

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

current→get_currentValue() YCurrent current→currentValue()YCurrent get_currentValue

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

```
YCurrent target get_currentValue
```

Returns :

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

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

current→get_highestValue() YCurrent
current→highestValue()YCurrent get_highestValue

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

```
YCurrent target get_highestValue
```

Returns :

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

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

current→get_logFrequency() YCurrent
current→logFrequency()YCurrent get_logFrequency

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

```
YCurrent target get_logFrequency
```

Returns :

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

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

current→get_logicalName() YCurrent
current→logicalName()YCurrent get_logicalName

Returns the logical name of the current sensor.

```
YCurrent target get_logicalName
```

Returns :

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

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

current→get_lowestValue() YCurrent current→lowestValue()YCurrent get_lowestValue

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

YCurrent **target** **get_lowestValue**

Returns :

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

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

current→get_recordedData() YCurrent
current→recordedData()YCurrent get_recordedData

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

```
YCurrent target get_recordedData startTime endTime
```

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

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

Parameters :

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

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

Returns :

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

current→get_reportFrequency() YCurrent
current→reportFrequency()YCurrent
get_reportFrequency

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

YCurrent **target** **get_reportFrequency**

Returns :

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

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

current→get_resolution() YCurrent
current→resolution()YCurrent get_resolution

Returns the resolution of the measured values.

```
YCurrent target get_resolution
```

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

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

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

current→get_unit() YCurrent
current→unit()YCurrent get_unit

Returns the measuring unit for the current.

```
YCurrent target get_unit
```

Returns :

a string corresponding to the measuring unit for the current

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

current→loadCalibrationPoints()YCurrent loadCalibrationPoints YCurrent

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

YCurrent **target loadCalibrationPoints rawValues refValues**

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

current→set_highestValue() YCurrent
current→setHighestValue()YCurrent
set_highestValue

Changes the recorded maximal value observed.

```
YCurrent target set_highestValue newval
```

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

current→set_logFrequency() YCurrent
current→setLogFrequency()YCurrent
set_logFrequency

Changes the datalogger recording frequency for this function.

```
YCurrent target set_logFrequency newval
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

current→set_logicalName() YCurrent
current→setLogicalName()YCurrent set_logicalName

Changes the logical name of the current sensor.

```
YCurrent 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 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()YCurrent set_lowestValue

Changes the recorded minimal value observed.

```
YCurrent target set_lowestValue newval
```

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

current→set_reportFrequency() YCurrent
current→setReportFrequency()YCurrent
set_reportFrequency

Changes the timed value notification frequency for this function.

```
YCurrent target set_reportFrequency newval
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

current→set_resolution() YCurrent
current→setResolution()YCurrent set_resolution

Changes the resolution of the measured physical values.

```
YCurrent target set_resolution newval
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

3.9. DataLogger function interface

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

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

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

Global functions
yFindDataLogger(**func**) Retrieves a data logger for a given identifier.
yFirstDataLogger() Starts the enumeration of data loggers currently accessible.
`YDataLogger` methods
datalogger→describe() Returns a short text that describes unambiguously the instance of the data logger in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
datalogger→forgetAllDataStreams() Clears the data logger memory and discards all recorded data streams.
datalogger→get_advertisedValue() Returns the current value of the data logger (no more than 6 characters).
datalogger→get_autoStart() Returns the default activation state of the data logger on power up.
datalogger→get_beaconDriven() Return true if the data logger is synchronised with the localization beacon.
datalogger→get_currentRunIndex() Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.
datalogger→get_dataSets() Returns a list of YDataSet objects that can be used to retrieve all measures stored by the data logger.
datalogger→get_dataStreams(**v**) Builds a list of all data streams hold by the data logger (legacy method).
datalogger→get_errorMessage() Returns the error message of the latest error with the data logger.
datalogger→get_errorType() Returns the numerical error code of the latest error with the data logger.
datalogger→get_friendlyName()

Returns a global identifier of the data logger in the format MODULE_NAME.FUNCTION_NAME.

datalogger→get_functionDescriptor()

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

datalogger→get_functionId()

Returns the hardware identifier of the data logger, without reference to the module.

datalogger→get_hardwareId()

Returns the unique hardware identifier of the data logger in the form SERIAL.FUNCTIONID.

datalogger→get_logicalName()

Returns the logical name of the data logger.

datalogger→get_module()

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

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

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

datalogger→get_recording()

Returns the current activation state of the data logger.

datalogger→get_timeUTC()

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

datalogger→get_userData()

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

datalogger→isOnline()

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

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

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

datalogger→load(**msValidity**)

Preloads the data logger cache with a specified validity duration.

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

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

datalogger→nextDataLogger()

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

datalogger→registerValueCallback(**callback**)

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

datalogger→set_autoStart(**newval**)

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

datalogger→set_beaconDriven(**newval**)

Changes the type of synchronisation of the data logger.

datalogger→set_logicalName(**newval**)

Changes the logical name of the data logger.

datalogger→set_recording(**newval**)

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

datalogger→set_timeUTC(**newval**)

Changes the current UTC time reference used for recorded data.

datalogger→set_userData(**data**)

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

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

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

datalogger→forgetAllDataStreams()YDataLogger forgetAllDataStreams

YDataLogger

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

YDataLogger **target** **forgetAllDataStreams**

This method also resets the current run index to zero.

Returns :

YAPI_SUCCESS if the call succeeds.

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

datalogger→get_advertisedValue() YDataLogger
datalogger→advertisedValue()YDataLogger
get_advertisedValue

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

YDataLogger **target get_advertisedValue**

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

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

datalogger→get_autoStart() YDataLogger
datalogger→autoStart()YDataLogger get_autoStart

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

YDataLogger **target** **get_autoStart**

Returns :

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

On failure, throws an exception or returns Y_AUTOSTART_INVALID.

datalogger→get_beaconDriven() YDataLogger
datalogger→beaconDriven()YDataLogger
get_beaconDriven

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

YDataLogger **target get_beaconDriven**

Returns :

either `Y_BEACONDRIVEN_OFF` or `Y_BEACONDRIVEN_ON`

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

datalogger→get_currentRunIndex() YDataLogger
datalogger→currentRunIndex()YDataLogger
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.

YDataLogger **target get_currentRunIndex**

Returns :

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

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

datalogger→get_dataSets() YDataLogger
datalogger→dataSets()YDataLogger get_dataSets

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

```
YDataLogger target get_dataSets
```

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

Returns :
a list of YDataSet object.

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

datalogger→get_logicalName() YDataLogger
datalogger→logicalName()YDataLogger
get_logicalName

Returns the logical name of the data logger.

YDataLogger **target** **get_logicalName**

Returns :
a string corresponding to the logical name of the data logger.

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

datalogger→get_recording() YDataLogger
datalogger→recording()YDataLogger get_recording

Returns the current activation state of the data logger.

YDataLogger **target get_recording**

Returns :

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

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

datalogger→get_timeUTC() YDataLogger
datalogger→timeUTC()YDataLogger get_timeUTC

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

YDataLogger **target get_timeUTC**

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

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

datalogger→set_autoStart() YDataLogger
datalogger→setAutoStart()YDataLogger
set_autoStart

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

YDataLogger **target set_autoStart newval**

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

datalogger→set_beaconDriven() YDataLogger
datalogger→setBeaconDriven()YDataLogger
set_beaconDriven

Changes the type of synchronisation of the data logger.

```
YDataLogger target set_beaconDriven newval
```

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

Parameters :

newval either `Y_BEACONDRIVEN_OFF` or `Y_BEACONDRIVEN_ON`, according to the type of synchronisation of the data logger

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

datalogger→set_logicalName() YDataLogger
datalogger→setLogicalName()YDataLogger
set_logicalName

Changes the logical name of the data logger.

```
YDataLogger 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 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()YDataLogger
set_recording

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

YDataLogger **target** **set_recording** **newval**

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

datalogger→set_timeUTC() YDataLogger
datalogger→setTimeUTC()YDataLogger set_timeUTC

Changes the current UTC time reference used for recorded data.

YDataLogger **target set_timeUTC newval**

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

3.10. Formatted data sequence

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

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

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

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

datarun→get_startTimeUTC() YDataRun
datarun→startTimeUTC()

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

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

Returns :

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

3.11. Recorded data sequence

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

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

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

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

```
js      <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 *
```

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.

3.12. Unformatted data sequence

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

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

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

```
js      <script type='text/javascript' src='yocto_api.js'></script>
nodejs  var yoctolib = require('yoctolib');
        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.

3.13. Digital IO function interface

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

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

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

Global functions
yFindDigitalIO(**func**) Retrieves a digital IO port for a given identifier.
yFirstDigitalIO() Starts the enumeration of digital IO ports currently accessible.
`YDigitalIO` methods
digitalio→delayedPulse(**bitno**, **ms_delay**, **ms_duration**) Schedules a pulse on a single bit for a specified duration.
digitalio→describe() Returns a short text that describes unambiguously the instance of the digital IO port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
digitalio→get_advertisedValue() Returns the current value of the digital IO port (no more than 6 characters).
digitalio→get_bitDirection(**bitno**) Returns the direction of a single bit from the I/O port (0 means the bit is an input, 1 an output).
digitalio→get_bitOpenDrain(**bitno**) Returns the type of electrical interface of a single bit from the I/O port.
digitalio→get_bitPolarity(**bitno**) Returns the polarity of a single bit from the I/O port (0 means the I/O works in regular mode, 1 means the I/O works in reverse mode).
digitalio→get_bitState(**bitno**) Returns the state of a single bit of the I/O port.
digitalio→get_errorMessage() Returns the error message of the latest error with the digital IO port.
digitalio→get_errorType() Returns the numerical error code of the latest error with the digital IO port.
digitalio→get_friendlyName() Returns a global identifier of the digital IO port in the format `MODULE_NAME.FUNCTION_NAME`.

digitalio→get_functionDescriptor()

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

digitalio→get_functionId()

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

digitalio→get_hardwareId()

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

digitalio→get_logicalName()

Returns the logical name of the digital IO port.

digitalio→get_module()

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

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

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

digitalio→get_outputVoltage()

Returns the voltage source used to drive output bits.

digitalio→get_portDirection()

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

digitalio→get_portOpenDrain()

Returns the electrical interface for each bit of the port.

digitalio→get_portPolarity()

Returns the polarity of all the bits of the port.

digitalio→get_portSize()

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

digitalio→get_portState()

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

digitalio→get_userData()

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

digitalio→isOnline()

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

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

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

digitalio→load(**msValidity**)

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

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

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

digitalio→nextDigitalIO()

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

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

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

digitalio→registerValueCallback(**callback**)

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

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

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

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

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

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

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

digitalio→set_bitState(bitno, bitstate)

Sets a single bit of the I/O port.

digitalio→set_logicalName(newval)

Changes the logical name of the digital IO port.

digitalio→set_outputVoltage(newval)

Changes the voltage source used to drive output bits.

digitalio→set_portDirection(newval)

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

digitalio→set_portOpenDrain(newval)

Changes the electrical interface for each bit of the port.

digitalio→set_portPolarity(newval)

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

digitalio→set_portState(newval)

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

digitalio→set_userData(data)

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

digitalio→toggle_bitState(bitno)

Reverts a single bit of the I/O port.

digitalio→wait_async(callback, context)

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

digitalio→delayedPulse()YDigitalIO delayedPulse YDigitalIO

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

YDigitalIO **target delayedPulse bitno ms_delay ms_duration**

The specified bit will be turned to 1, and then back to 0 after the given duration.

Parameters :

bitno	the bit number; lowest bit has index 0
ms_delay	waiting time before the pulse, in milliseconds
ms_duration	desired pulse duration in milliseconds. Be aware that the device time resolution is not guaranteed up to the millisecond.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

digitalio→get_advertisedValue() YDigitalIO
digitalio→advertisedValue()YDigitalIO
get_advertisedValue

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

YDigitalIO **target get_advertisedValue**

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

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

digitalio→get_bitDirection() YDigitalIO
digitalio→bitDirection()YDigitalIO get_bitDirection

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

YDigitalIO **target** **get_bitDirection** **bitno**

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

YAPI_SUCCESS if the call succeeds.

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

digitalio→get_bitOpenDrain() YDigitalIO
digitalio→bitOpenDrain()YDigitalIO get_bitOpenDrain

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

```
YDigitalIO target get_bitOpenDrain bitno
```

(0 means the bit is an input, 1 an output).

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

0 means the a bit is a regular input/output, 1 means the bit is an open-drain (open-collector) input/output.

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

digitalio→get_bitPolarity() YDigitalIO
digitalio→bitPolarity()YDigitalIO 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).

YDigitalIO **target get_bitPolarity bitno**

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

YAPI_SUCCESS if the call succeeds.

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

digitalio→get_bitState() YDigitalIO
digitalio→bitState()YDigitalIO get_bitState

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

YDigitalIO **target get_bitState bitno**

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

the bit state (0 or 1)

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

digitalio→get_logicalName() YDigitalIO
digitalio→logicalName()YDigitalIO get_logicalName

Returns the logical name of the digital IO port.

YDigitalIO **target** **get_logicalName**

Returns :
a string corresponding to the logical name of the digital IO port.

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

digitalio→get_outputVoltage() YDigitalIO
digitalio→outputVoltage()YDigitalIO
get_outputVoltage

Returns the voltage source used to drive output bits.

YDigitalIO **target get_outputVoltage**

Returns :

a value among `Y_OUTPUTVOLTAGE_USB_5V`, `Y_OUTPUTVOLTAGE_USB_3V` and `Y_OUTPUTVOLTAGE_EXT_V` corresponding to the voltage source used to drive output bits

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

digitalio→get_portDirection() YDigitalIO
digitalio→portDirection()YDigitalIO get_portDirection

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

YDigitalIO **target** **get_portDirection**

Returns :

an integer corresponding to the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output

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

digitalio→get_portOpenDrain() YDigitalIO
digitalio→portOpenDrain()YDigitalIO
get_portOpenDrain

Returns the electrical interface for each bit of the port.

YDigitalIO **target get_portOpenDrain**

For each bit set to 0 the matching I/O works in the regular, intuitive way, for each bit set to 1, the I/O works in reverse mode.

Returns :
an integer corresponding to the electrical interface for each bit of the port

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

digitalio→get_portPolarity() YDigitalIO
digitalio→portPolarity()YDigitalIO get_portPolarity

Returns the polarity of all the bits of the port.

YDigitalIO **target** **get_portPolarity**

For each bit set to 0, the matching I/O works the regular, intuitive way; for each bit set to 1, the I/O works in reverse mode.

Returns :
an integer corresponding to the polarity of all the bits of the port

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

digitalio→get_portSize() YDigitalIO
digitalio→portSize()YDigitalIO get_portSize

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

YDigitalIO **target get_portSize**

Returns :
an integer corresponding to the number of bits implemented in the I/O port

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

digitalio→get_portState() YDigitalIO
digitalio→portState()YDigitalIO get_portState

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

YDigitalIO **target** **get_portState**

Returns :
an integer corresponding to the digital IO port state: bit 0 represents input 0, and so on

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

digitalio→pulse()YDigitalIO pulse YDigitalIO

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

```
YDigitalIO target pulse bitno ms_duration
```

The specified bit will be turned to 1, and then back to 0 after the given duration.

Parameters :

bitno	the bit number; lowest bit has index 0
ms_duration	desired pulse duration in milliseconds. Be aware that the device time resolution is not guaranteed up to the millisecond.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

digitalio→set_bitDirection() YDigitalIO
digitalio→setBitDirection()YDigitalIO set_bitDirection

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

YDigitalIO **target set_bitDirection bitno bitdirection**

Parameters :

bitno the bit number; lowest bit has index 0

bitdirection direction to set, 0 makes the bit an input, 1 makes it an output. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

digitalio→set_bitOpenDrain() YDigitalIO
digitalio→setBitOpenDrain()YDigitalIO
set_bitOpenDrain

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

YDigitalIO **target set_bitOpenDrain bitno opendrain**

Parameters :

bitno the bit number; lowest bit has index 0

opendrain 0 makes a bit a regular input/output, 1 makes it an open-drain (open-collector) input/output. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

digitalio→set_bitPolarity() YDigitalIO
digitalio→setBitPolarity()YDigitalIO set_bitPolarity

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

YDigitalIO **target set_bitPolarity bitno bitpolarity**

Parameters :

bitno	the bit number; lowest bit has index 0.
bitpolarity	polarity to set, 0 makes the I/O work in regular mode, 1 makes the I/O works in reverse mode. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

digitalio→set_bitState() YDigitalIO
digitalio→setBitState()YDigitalIO set_bitState

Sets a single bit of the I/O port.

```
YDigitalIO target set_bitState bitno bitstate
```

Parameters :

bitno the bit number; lowest bit has index 0

bitstate the state of the bit (1 or 0)

Returns :

YAPI_SUCCESS if the call succeeds.

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

digitalio→set_logicalName() YDigitalIO
digitalio→setLogicalName()YDigitalIO
set_logicalName

Changes the logical name of the digital IO port.

YDigitalIO **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 digital IO port.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

digitalio→set_outputVoltage() YDigitalIO
digitalio→setOutputVoltage()YDigitalIO
set_outputVoltage

Changes the voltage source used to drive output bits.

```
YDigitalIO target set_outputVoltage newval
```

Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Parameters :

newval a value among `Y_OUTPUTVOLTAGE_USB_5V`, `Y_OUTPUTVOLTAGE_USB_3V` and `Y_OUTPUTVOLTAGE_EXT_V` corresponding to the voltage source used to drive output bits

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

digitalio→set_portDirection() YDigitalIO
digitalio→setPortDirection()YDigitalIO
set_portDirection

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

```
YDigitalIO target set_portDirection newval
```

Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Parameters :

newval an integer corresponding to the IO direction of all bits of the port: 0 makes a bit an input, 1 makes it an output

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

digitalio→set_portOpenDrain() YDigitalIO
digitalio→setPortOpenDrain()YDigitalIO
set_portOpenDrain

Changes the electrical interface for each bit of the port.

YDigitalIO **target set_portOpenDrain newval**

0 makes a bit a regular input/output, 1 makes it an open-drain (open-collector) input/output. Remember to call the `saveToFlash()` method to make sure the setting is kept after a reboot.

Parameters :

newval an integer corresponding to the electrical interface for each bit of the port

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

digitalio→set_portPolarity() YDigitalIO
digitalio→setPortPolarity()YDigitalIO set_portPolarity

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

```
YDigitalIO target set_portPolarity newval
```

Remember to call the `saveToFlash()` method to make sure the setting will be kept after a reboot.

Parameters :

newval an integer corresponding to the polarity of all the bits of the port: 0 makes a bit an input, 1 makes it an output

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

digitalio→set_portState() YDigitalIO
digitalio→setPortState()YDigitalIO set_portState

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

```
YDigitalIO target set_portState newval
```

This function has no effect on bits configured as input in `portDirection`.

Parameters :

newval an integer corresponding to the digital IO port state: bit 0 represents input 0, and so on

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

digitalio→toggle_bitState()YDigitalIO toggle_bitState YDigitalIO

Reverts a single bit of the I/O port.

```
YDigitalIO target toggle_bitState bitno
```

Parameters :

bitno the bit number; lowest bit has index 0

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

3.14. Display function interface

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

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

```
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');
cpp     #include "yocto_display.h"
m       #import "yocto_display.h"
pas     uses yocto_display;
vb      yocto_display.vb
cs      yocto_display.cs
java    import com.yoctopuce.YoctoAPI.YDisplay;
py      from yocto_display import *
```

Global functions
yFindDisplay(**func**) Retrieves a display for a given identifier.
yFirstDisplay() Starts the enumeration of displays currently accessible.
`YDisplay` methods
display→copyLayerContent(**srcLayerId**, **dstLayerId**) Copies the whole content of a layer to another layer.
display→describe() Returns a short text that describes unambiguously the instance of the display in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
display→fade(**brightness**, **duration**) Smoothly changes the brightness of the screen to produce a fade-in or fade-out effect.
display→get_advertisedValue() Returns the current value of the display (no more than 6 characters).
display→get_brightness() Returns the luminosity of the module informative leds (from 0 to 100).
display→get_displayHeight() Returns the display height, in pixels.
display→get_displayLayer(**layerId**) Returns a YDisplayLayer object that can be used to draw on the specified layer.
display→get_displayType() Returns the display type: monochrome, gray levels or full color.
display→get_displayWidth() Returns the display width, in pixels.
display→get_enabled() Returns true if the screen is powered, false otherwise.
display→get_errorMessage() Returns the error message of the latest error with the display.

display→get_errorType()
Returns the numerical error code of the latest error with the display.

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

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

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

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

display→get_layerCount()
Returns the number of available layers to draw on.

display→get_layerHeight()
Returns the height of the layers to draw on, in pixels.

display→get_layerWidth()
Returns the width of the layers to draw on, in pixels.

display→get_logicalName()
Returns the logical name of the display.

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

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

display→get_orientation()
Returns the currently selected display orientation.

display→get_startupSeq()
Returns the name of the sequence to play when the displayed is powered on.

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

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

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

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

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

display→newSequence()
Starts to record all display commands into a sequence, for later replay.

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

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

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

display→registerValueCallback(callback)

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

display→resetAll()

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

display→saveSequence(**sequenceName**)

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

display→set_brightness(**newval**)

Changes the brightness of the display.

display→set_enabled(**newval**)

Changes the power state of the display.

display→set_logicalName(**newval**)

Changes the logical name of the display.

display→set_orientation(**newval**)

Changes the display orientation.

display→set_startupSeq(**newval**)

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

display→set_userData(**data**)

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

display→stopSequence()

Stops immediately any ongoing sequence replay.

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

Swaps the whole content of two layers.

display→upload(**pathname**, **content**)

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

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

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

display→copyLayerContent()YDisplay copyLayerContent YDisplay

Copies the whole content of a layer to another layer.

```
YDisplay target copyLayerContent srcLayerId dstLayerId
```

The color and transparency of all the pixels from the destination layer are set to match the source pixels. This method only affects the displayed content, but does not change any property of the layer object. Note that layer 0 has no transparency support (it is always completely opaque).

Parameters :

srcLayerId the identifier of the source layer (a number in range 0..layerCount-1)

dstLayerId the identifier of the destination layer (a number in range 0..layerCount-1)

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

display→fade()YDisplay fade YDisplay

Smoothly changes the brightness of the screen to produce a fade-in or fade-out effect.

YDisplay **target fade brightness duration**

Parameters :

brightness the new screen brightness

duration duration of the brightness transition, in milliseconds.

Returns :

YAPI_SUCCESS if the call succeeds.

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

display→get_advertisedValue() YDisplay
display→advertisedValue()YDisplay
get_advertisedValue

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

YDisplay **target** **get_advertisedValue**

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

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

display→get_brightness() YDisplay
display→brightness()YDisplay get_brightness

Returns the luminosity of the module informative leds (from 0 to 100).

YDisplay **target get_brightness**

Returns :

an integer corresponding to the luminosity of the module informative leds (from 0 to 100)

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

display→get_displayHeight() YDisplay
display→displayHeight()YDisplay get_displayHeight

Returns the display height, in pixels.

YDisplay **target** **get_displayHeight**

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()YDisplay get_displayType

Returns the display type: monochrome, gray levels or full color.

YDisplay **target get_displayType**

Returns :

a value among `Y_DISPLAYTYPE_MONO`, `Y_DISPLAYTYPE_GRAY` and `Y_DISPLAYTYPE_RGB` corresponding to the display type: monochrome, gray levels or full color

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

display→get_displayWidth() YDisplay
display→displayWidth()YDisplay get_displayWidth

Returns the display width, in pixels.

YDisplay **target** **get_displayWidth**

Returns :
an integer corresponding to the display width, in pixels

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

display→get_enabled() YDisplay
display→enabled()YDisplay get_enabled

Returns true if the screen is powered, false otherwise.

YDisplay **target get_enabled**

Returns :

either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to true if the screen is powered, false otherwise

On failure, throws an exception or returns Y_ENABLED_INVALID.

display→get_layerCount() YDisplay
display→layerCount()YDisplay get_layerCount

Returns the number of available layers to draw on.

YDisplay **target** **get_layerCount**

Returns :
an integer corresponding to the number of available layers to draw on

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

display→get_layerHeight() YDisplay
display→layerHeight()YDisplay get_layerHeight

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

YDisplay **target get_layerHeight**

Returns :
an integer corresponding to the height of the layers to draw on, in pixels

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

display→get_layerWidth() YDisplay
display→layerWidth()YDisplay get_layerWidth

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

YDisplay **target** **get_layerWidth**

Returns :
an integer corresponding to the width of the layers to draw on, in pixels

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

display→get_logicalName() YDisplay
display→logicalName()YDisplay get_logicalName

Returns the logical name of the display.

YDisplay **target get_logicalName**

Returns :
a string corresponding to the logical name of the display.

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

display→get_orientation() YDisplay
display→orientation()YDisplay get_orientation

Returns the currently selected display orientation.

YDisplay **target** **get_orientation**

Returns :

a value among `Y_ORIENTATION_LEFT`, `Y_ORIENTATION_UP`, `Y_ORIENTATION_RIGHT` and `Y_ORIENTATION_DOWN` corresponding to the currently selected display orientation

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

display→get_startupSeq() YDisplay
display→startupSeq()YDisplay get_startupSeq

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

YDisplay **target get_startupSeq**

Returns :
a string corresponding to the name of the sequence to play when the displayed is powered on

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

display→newSequence()YDisplay newSequence YDisplay

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

YDisplay **target** **newSequence**

The name used to store the sequence is specified when calling `saveSequence()`, once the recording is complete.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

display→pauseSequence()YDisplay pauseSequence YDisplay

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

```
YDisplay target pauseSequence delay_ms
```

This method can be used while recording a display sequence, to insert a timed wait in the sequence (without any immediate effect). It can also be used dynamically while playing a pre-recorded sequence, to suspend or resume the execution of the sequence. To cancel a delay, call the same method with a zero delay.

Parameters :

delay_ms the duration to wait, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

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

display→playSequence()YDisplay playSequence YDisplay

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

YDisplay **target playSequence sequenceName**

Parameters :

sequenceName the name of the newly created sequence

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

display→resetAll()YDisplay resetAll YDisplay

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

```
YDisplay target resetAll
```

Using this function in a sequence will kill the sequence play-back. Don't use that function to reset the display at sequence start-up.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

display→saveSequence()YDisplay saveSequence YDisplay

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

```
YDisplay target saveSequence sequenceName
```

The sequence can be later replayed using `playSequence()`.

Parameters :

sequenceName the name of the newly created sequence

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

display→set_brightness() YDisplay
display→setBrightness()YDisplay set_brightness

Changes the brightness of the display.

```
YDisplay target set_brightness newval
```

The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer corresponding to the brightness of the display

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

display→set_enabled() YDisplay
display→setEnabled()YDisplay set_enabled

Changes the power state of the display.

YDisplay **target** **set_enabled** **newval**

Parameters :

newval either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to the power state of the display

Returns :

YAPI_SUCCESS if the call succeeds.

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

display→set_logicalName() YDisplay
display→setLogicalName()YDisplay set_logicalName

Changes the logical name of the display.

```
YDisplay 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 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()YDisplay set_orientation

Changes the display orientation.

YDisplay **target** **set_orientation** **newval**

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

Parameters :

newval a value among `Y_ORIENTATION_LEFT`, `Y_ORIENTATION_UP`, `Y_ORIENTATION_RIGHT` and `Y_ORIENTATION_DOWN` corresponding to the display orientation

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

display→set_startupSeq() YDisplay
display→setStartupSeq()YDisplay set_startupSeq

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

YDisplay **target set_startupSeq newval**

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

Parameters :

newval a string corresponding to the name of the sequence to play when the displayed is powered on

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

display→stopSequence()YDisplay stopSequence YDisplay

Stops immediately any ongoing sequence replay.

YDisplay **target stopSequence**

The display is left as is.

Returns :
`YAPI_SUCCESS` if the call succeeds.

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

display→swapLayerContent()YDisplay swapLayerContent YDisplay

Swaps the whole content of two layers.

```
YDisplay target swapLayerContent layerIdA layerIdB
```

The color and transparency of all the pixels from the two layers are swapped. This method only affects the displayed content, but does not change any property of the layer objects. In particular, the visibility of each layer stays unchanged. When used between onae 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()YDisplay upload YDisplay

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

YDisplay **target upload pathname content**

If a file already exists with the same path name, its content is overwritten.

Parameters :

pathname path and name of the new file to create

content binary buffer with the content to set

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

3.15. DisplayLayer object interface

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

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

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

YDisplayLayer methods

displaylayer→clear()

Erases the whole content of the layer (makes it fully transparent).

displaylayer→clearConsole()

Blanks the console area within console margins, and resets the console pointer to the upper left corner of the console.

displaylayer→consoleOut(**text**)

Outputs a message in the console area, and advances the console pointer accordingly.

displaylayer→drawBar(**x1**, **y1**, **x2**, **y2**)

Draws a filled rectangular bar at a specified position.

displaylayer→drawBitmap(**x**, **y**, **w**, **bitmap**, **bgcol**)

Draws a bitmap at the specified position.

displaylayer→drawCircle(**x**, **y**, **r**)

Draws an empty circle at a specified position.

displaylayer→drawDisc(**x**, **y**, **r**)

Draws a filled disc at a given position.

displaylayer→drawImage(**x**, **y**, **imagename**)

Draws a GIF image at the specified position.

displaylayer→drawPixel(**x**, **y**)

Draws a single pixel at the specified position.

displaylayer→drawRect(**x1**, **y1**, **x2**, **y2**)

Draws an empty rectangle at a specified position.

displaylayer→drawText(**x**, **y**, **anchor**, **text**)

Draws a text string at the specified position.

displaylayer→get_display()

Gets parent YDisplay.

displaylayer→get_displayHeight()

Returns the display height, in pixels.

displaylayer→get_displayWidth()

Returns the display width, in pixels.

displaylayer→get_layerHeight()

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

displaylayer→get_layerWidth()

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

displaylayer→hide()

Hides the layer.

displaylayer→lineTo(**x**, **y**)

Draws a line from current drawing pointer position to the specified position.

displaylayer→moveTo(**x**, **y**)

Moves the drawing pointer of this layer to the specified position.

displaylayer→reset()

Reverts the layer to its initial state (fully transparent, default settings).

displaylayer→selectColorPen(**color**)

Selects the pen color for all subsequent drawing functions, including text drawing.

displaylayer→selectEraser()

Selects an eraser instead of a pen for all subsequent drawing functions, except for bitmap copy functions.

displaylayer→selectFont(**fontname**)

Selects a font to use for the next text drawing functions, by providing the name of the font file.

displaylayer→selectGrayPen(**graylevel**)

Selects the pen gray level for all subsequent drawing functions, including text drawing.

displaylayer→setAntialiasingMode(**mode**)

Enables or disables anti-aliasing for drawing oblique lines and circles.

displaylayer→setConsoleBackground(**bgcol**)

Sets up the background color used by the `clearConsole` function and by the console scrolling feature.

displaylayer→setConsoleMargins(**x1**, **y1**, **x2**, **y2**)

Sets up display margins for the `consoleOut` function.

displaylayer→setConsoleWordWrap(**wordwrap**)

Sets up the wrapping behaviour used by the `consoleOut` function.

displaylayer→setLayerPosition(**x**, **y**, **scrollTime**)

Sets the position of the layer relative to the display upper left corner.

displaylayer→unhide()

Shows the layer.

displaylayer→clear()YDisplayLayer clear YDisplayLayer

Erases the whole content of the layer (makes it fully transparent).

YDisplay **target** [-layer **layerId**] **clear**

This method does not change any other attribute of the layer. To reinitialize the layer attributes to defaults settings, use the method `reset()` instead.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

displaylayer→clearConsole()YDisplayLayer clearConsole YDisplayLayer

Blanks the console area within console margins, and resets the console pointer to the upper left corner of the console.

YDisplay **target** [-layer **layerId**] **clearConsole**

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

displaylayer→consoleOut()YDisplayLayer consoleOut — YDisplayLayer

Outputs a message in the console area, and advances the console pointer accordingly.

YDisplay **target** [-layer **layerId**] **consoleOut** **text**

The console pointer position is automatically moved to the beginning of the next line when a newline character is met, or when the right margin is hit. When the new text to display extends below the lower margin, the console area is automatically scrolled up.

Parameters :

text the message to display

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

displaylayer→drawBar()YDisplayLayer drawBar YDisplayLayer

Draws a filled rectangular bar at a specified position.

YDisplay **target** [-layer **layerId**] **drawBar** **x1 y1 x2 y2**

Parameters :

x1 the distance from left of layer to the left border of the rectangle, in pixels
y1 the distance from top of layer to the top border of the rectangle, in pixels
x2 the distance from left of layer to the right border of the rectangle, in pixels
y2 the distance from top of layer to the bottom border of the rectangle, in pixels

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

displaylayer→drawBitmap()YDisplayLayer drawBitmap YDisplayLayer

Draws a bitmap at the specified position.

YDisplay **target** [-layer **layerId**] **drawBitmap** **x y w bitmap bgcol**

The bitmap is provided as a binary object, where each pixel maps to a bit, from left to right and from top to bottom. The most significant bit of each byte maps to the leftmost pixel, and the least significant bit maps to the rightmost pixel. Bits set to 1 are drawn using the layer selected pen color. Bits set to 0 are drawn using the specified background gray level, unless -1 is specified, in which case they are not drawn at all (as if transparent).

Parameters :

x	the distance from left of layer to the left of the bitmap, in pixels
y	the distance from top of layer to the top of the bitmap, in pixels
w	the width of the bitmap, in pixels
bitmap	a binary object
bgcol	the background gray level to use for zero bits (0 = black, 255 = white), or -1 to leave the pixels unchanged

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

displaylayer→drawCircle()YDisplayLayer drawCircle YDisplayLayer

Draws an empty circle at a specified position.

YDisplay **target** [-layer **layerId**] **drawCircle** **x y r**

Parameters :

x the distance from left of layer to the center of the circle, in pixels
y the distance from top of layer to the center of the circle, in pixels
r the radius of the circle, in pixels

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

displaylayer→drawDisc()YDisplayLayer drawDisc YDisplayLayer

Draws a filled disc at a given position.

YDisplay **target** [-layer **layerId**] **drawDisc** **x y r**

Parameters :

x the distance from left of layer to the center of the disc, in pixels
y the distance from top of layer to the center of the disc, in pixels
r the radius of the disc, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

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

displaylayer→drawImage()YDisplayLayer drawImage YDisplayLayer

Draws a GIF image at the specified position.

YDisplay **target** [-layer **layerId**] **drawImage** **x y imagename**

The GIF image must have been previously uploaded to the device built-in memory. If you experience problems using an image file, check the device logs for any error message such as missing image file or bad image file format.

Parameters :

x	the distance from left of layer to the left of the image, in pixels
y	the distance from top of layer to the top of the image, in pixels
imagename	the GIF file name

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

displaylayer→drawPixel()YDisplayLayer drawPixel YDisplayLayer

Draws a single pixel at the specified position.

YDisplay **target** [-layer **layerId**] **drawPixel** **x y**

Parameters :

x the distance from left of layer, in pixels

y the distance from top of layer, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

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

displaylayer→drawRect()YDisplayLayer drawRect YDisplayLayer

Draws an empty rectangle at a specified position.

YDisplay **target** [-layer **layerId**] **drawRect** **x1 y1 x2 y2**

Parameters :

x1 the distance from left of layer to the left border of the rectangle, in pixels
y1 the distance from top of layer to the top border of the rectangle, in pixels
x2 the distance from left of layer to the right border of the rectangle, in pixels
y2 the distance from top of layer to the bottom border of the rectangle, in pixels

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

displaylayer→drawText()YDisplayLayer drawText YDisplayLayer

Draws a text string at the specified position.

YDisplay **target** [-layer **layerId**] **drawText x y anchor text**

The point of the text that is aligned to the specified pixel position is called the anchor point, and can be chosen among several options. Text is rendered from left to right, without implicit wrapping.

Parameters :

x	the distance from left of layer to the text anchor point, in pixels
y	the distance from top of layer to the text anchor point, in pixels
anchor	the text anchor point, chosen among the `Y_ALIGN` enumeration: `Y_ALIGN_TOP_LEFT`, `Y_ALIGN_CENTER_LEFT`, `Y_ALIGN_BASELINE_LEFT`, `Y_ALIGN_BOTTOM_LEFT`, `Y_ALIGN_TOP_CENTER`, `Y_ALIGN_CENTER`, `Y_ALIGN_BASELINE_CENTER`, `Y_ALIGN_BOTTOM_CENTER`, `Y_ALIGN_TOP_DECIMAL`, `Y_ALIGN_CENTER_DECIMAL`, `Y_ALIGN_BASELINE_DECIMAL`, `Y_ALIGN_BOTTOM_DECIMAL`, `Y_ALIGN_TOP_RIGHT`, `Y_ALIGN_CENTER_RIGHT`, `Y_ALIGN_BASELINE_RIGHT`, `Y_ALIGN_BOTTOM_RIGHT`.
text	the text string to draw

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

displaylayer→get_displayHeight() YDisplayLayer
displaylayer→displayHeight()YDisplayLayer
get_displayHeight

Returns the display height, in pixels.

YDisplay **target** [-layer **layerId**] **get_displayHeight**

Returns :

an integer corresponding to the display height, in pixels On failure, throws an exception or returns Y_DISPLAYHEIGHT_INVALID.

displaylayer→get_displayWidth() YDisplayLayer
displaylayer→displayWidth()YDisplayLayer
get_displayWidth

Returns the display width, in pixels.

YDisplay **target** [-layer **layerId**] **get_displayWidth**

Returns :

an integer corresponding to the display width, in pixels On failure, throws an exception or returns Y_DISPLAYWIDTH_INVALID.

displaylayer→get_layerHeight() YDisplayLayer
displaylayer→layerHeight()YDisplayLayer
get_layerHeight

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

YDisplay **target** [-layer **layerId**] **get_layerHeight**

Returns :
an integer corresponding to the height of the layers to draw on, in pixels

On failure, throws an exception or returns Y_LAYERHEIGHT_INVALID.

displaylayer→get_layerWidth() YDisplayLayer
displaylayer→layerWidth()YDisplayLayer
get_layerWidth

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

YDisplay **target** [-layer **layerId**] **get_layerWidth**

Returns :
an integer corresponding to the width of the layers to draw on, in pixels

On failure, throws an exception or returns Y_LAYERWIDTH_INVALID.

displaylayer→hide()YDisplayLayer hide YDisplayLayer

Hides the layer.

```
YDisplay target [-layer layerId] hide
```

The state of the layer is perserved but the layer is not displayed on the screen until the next call to `unhide()`. Hiding the layer can positively affect the drawing speed, since it postpones the rendering until all operations are completed (double-buffering).

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

displaylayer→lineTo()YDisplayLayer lineTo YDisplayLayer

Draws a line from current drawing pointer position to the specified position.

YDisplay **target** [-layer **layerId**] **lineTo** **x y**

The specified destination pixel is included in the line. The pointer position is then moved to the end point of the line.

Parameters :

x the distance from left of layer to the end point of the line, in pixels

y the distance from top of layer to the end point of the line, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

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

displaylayer→moveTo()YDisplayLayer moveTo YDisplayLayer

Moves the drawing pointer of this layer to the specified position.

YDisplay **target** [-layer **layerId**] **moveTo** **x y**

Parameters :

x the distance from left of layer, in pixels

y the distance from top of layer, in pixels

Returns :

YAPI_SUCCESS if the call succeeds.

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

displaylayer→reset()YDisplayLayer reset — YDisplayLayer

Reverts the layer to its initial state (fully transparent, default settings).

YDisplay **target** [-layer **layerId**] **reset**

Reinitializes the drawing pointer to the upper left position, and selects the most visible pen color. If you only want to erase the layer content, use the method `clear()` instead.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

displaylayer→selectColorPen()YDisplayLayer selectColorPen YDisplayLayer

Selects the pen color for all subsequent drawing functions, including text drawing.

YDisplay **target** [-layer **layerId**] **selectColorPen** **color**

The pen color is provided as an RGB value. For grayscale or monochrome displays, the value is automatically converted to the proper range.

Parameters :

color the desired pen color, as a 24-bit RGB value

Returns :

YAPI_SUCCESS if the call succeeds.

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

displaylayer→selectEraser()YDisplayLayer selectEraser YDisplayLayer

Selects an eraser instead of a pen for all subsequent drawing functions, except for bitmap copy functions.

YDisplay **target** [-layer **layerId**] **selectEraser**

Any point drawn using the eraser becomes transparent (as when the layer is empty), showing the other layers beneath it.

Returns :

YAPI_SUCCESS if the call succeeds.

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

displaylayer→selectFont()YDisplayLayer selectFont YDisplayLayer

Selects a font to use for the next text drawing functions, by providing the name of the font file.

YDisplay **target** [-layer **layerId**] **selectFont** **fontname**

You can use a built-in font as well as a font file that you have previously uploaded to the device built-in memory. If you experience problems selecting a font file, check the device logs for any error message such as missing font file or bad font file format.

Parameters :

fontname the font file name

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

displaylayer→selectGrayPen()YDisplayLayer selectGrayPen YDisplayLayer

Selects the pen gray level for all subsequent drawing functions, including text drawing.

YDisplay **target** [-layer **layerId**] **selectGrayPen** **graylevel**

The gray level is provided as a number between 0 (black) and 255 (white, or whichever the lighest 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()YDisplayLayer setAntialiasingMode YDisplayLayer

Enables or disables anti-aliasing for drawing oblique lines and circles.

YDisplay **target** [-layer **layerId**] **setAntialiasingMode** **mode**

Anti-aliasing provides a smoother aspect when looked from far enough, but it can add fuzzyness when the display is looked from very close. At the end of the day, it is your personal choice. Anti-aliasing is enabled by default on grayscale and color displays, but you can disable it if you prefer. This setting has no effect on monochrome displays.

Parameters :

mode true to enable antialiasing, false to disable it.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

displaylayer→setConsoleBackground() YDisplayLayer
YDisplayLayer setConsoleBackground

Sets up the background color used by the `clearConsole` function and by the console scrolling feature.

YDisplay **target** [-layer **layerId**] **setConsoleBackground** **bgcol**

Parameters :

bgcol the background gray level to use when scrolling (0 = black, 255 = white), or -1 for transparent

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

displaylayer→setConsoleMargins()YDisplayLayer setConsoleMargins YDisplayLayer

Sets up display margins for the `consoleOut` function.

YDisplay **target** [-layer **layerId**] **setConsoleMargins** **x1 y1 x2 y2**

Parameters :

x1 the distance from left of layer to the left margin, in pixels
y1 the distance from top of layer to the top margin, in pixels
x2 the distance from left of layer to the right margin, in pixels
y2 the distance from top of layer to the bottom margin, in pixels

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

displaylayer→setConsoleWordWrap()YDisplayLayer setConsoleWordWrap

YDisplayLayer

Sets up the wrapping behaviour used by the `consoleOut` function.

YDisplay **target** [-layer **layerId**] **setConsoleWordWrap** **wordwrap**

Parameters :

wordwrap `true` to wrap only between words, `false` to wrap on the last column anyway.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

displaylayer→setLayerPosition()YDisplayLayer setLayerPosition YDisplayLayer

Sets the position of the layer relative to the display upper left corner.

YDisplay **target** [-layer **layerId**] **setLayerPosition** **x y scrollTime**

When smooth scrolling is used, the display offset of the layer is automatically updated during the next milliseconds to animate the move of the layer.

Parameters :

x	the distance from left of display to the upper left corner of the layer
y	the distance from top of display to the upper left corner of the layer
scrollTime	number of milliseconds to use for smooth scrolling, or 0 if the scrolling should be immediate.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

displaylayer→unhide()YDisplayLayer unhide YDisplayLayer

Shows the layer.

YDisplay **target** [-layer **layerId**] **unhide**

Shows the layer again after a hide command.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

3.16. External power supply control interface

Yoctopuce application programming interface allows you to control the power source to use for module functions that require high current. The module can also automatically disconnect the external power when a voltage drop is observed on the external power source (external battery running out of power).

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_dualpower.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YDualPower = yoctolib.YDualPower;
php     require_once('yocto_dualpower.php');
cpp     #include "yocto_dualpower.h"
m       #import "yocto_dualpower.h"
pas     uses yocto_dualpower;
vb      yocto_dualpower.vb
cs      yocto_dualpower.cs
java    import com.yoctopuce.YoctoAPI.YDualPower;
py      from yocto_dualpower import *
```

Global functions
yFindDualPower(**func**) Retrieves a dual power control for a given identifier.
yFirstDualPower() Starts the enumeration of dual power controls currently accessible.
`YDualPower` methods
dualpower→describe() Returns a short text that describes unambiguously the instance of the power control in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
dualpower→get_advertisedValue() Returns the current value of the power control (no more than 6 characters).
dualpower→get_errorMessage() Returns the error message of the latest error with the power control.
dualpower→get_errorType() Returns the numerical error code of the latest error with the power control.
dualpower→get_extVoltage() Returns the measured voltage on the external power source, in millivolts.
dualpower→get_friendlyName() Returns a global identifier of the power control in the format `MODULE_NAME.FUNCTION_NAME`.
dualpower→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
dualpower→get_functionId() Returns the hardware identifier of the power control, without reference to the module.
dualpower→get_hardwareId() Returns the unique hardware identifier of the power control in the form `SERIAL.FUNCTIONID`.
dualpower→get_logicalName() Returns the logical name of the power control.
dualpower→get_module()

Gets the YModule object for the device on which the function is located.

dualpower→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

dualpower→get_powerControl()

Returns the selected power source for module functions that require lots of current.

dualpower→get_powerState()

Returns the current power source for module functions that require lots of current.

dualpower→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

dualpower→isOnline()

Checks if the power control is currently reachable, without raising any error.

dualpower→isOnline_async(callback, context)

Checks if the power control is currently reachable, without raising any error (asynchronous version).

dualpower→load(msValidity)

Preloads the power control cache with a specified validity duration.

dualpower→load_async(msValidity, callback, context)

Preloads the power control cache with a specified validity duration (asynchronous version).

dualpower→nextDualPower()

Continues the enumeration of dual power controls started using yFirstDualPower().

dualpower→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

dualpower→set_logicalName(newval)

Changes the logical name of the power control.

dualpower→set_powerControl(newval)

Changes the selected power source for module functions that require lots of current.

dualpower→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

dualpower→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

dualpower→get_advertisedValue() YDualPower
dualpower→advertisedValue()YDualPower
get_advertisedValue

Returns the current value of the power control (no more than 6 characters).

YDualPower **target** **get_advertisedValue**

Returns :
a string corresponding to the current value of the power control (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

dualpower→get_extVoltage() YDualPower
dualpower→extVoltage()YDualPower get_extVoltage

Returns the measured voltage on the external power source, in millivolts.

YDualPower **target get_extVoltage**

Returns :
an integer corresponding to the measured voltage on the external power source, in millivolts

On failure, throws an exception or returns `Y_EXTVOLTAGE_INVALID`.

dualpower→get_logicalName() YDualPower
dualpower→logicalName()YDualPower
get_logicalName

Returns the logical name of the power control.

```
YDualPower target get_logicalName
```

Returns :
a string corresponding to the logical name of the power control.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

dualpower→get_powerControl() YDualPower
dualpower→powerControl()YDualPower
get_powerControl

Returns the selected power source for module functions that require lots of current.

YDualPower **target get_powerControl**

Returns :

a value among `Y_POWERCONTROL_AUTO`, `Y_POWERCONTROL_FROM_USB`, `Y_POWERCONTROL_FROM_EXT` and `Y_POWERCONTROL_OFF` corresponding to the selected power source for module functions that require lots of current

On failure, throws an exception or returns `Y_POWERCONTROL_INVALID`.

dualpower→get_powerState() YDualPower
dualpower→powerState()YDualPower
get_powerState

Returns the current power source for module functions that require lots of current.

YDualPower **target** **get_powerState**

Returns :

a value among `Y_POWERSTATE_OFF`, `Y_POWERSTATE_FROM_USB` and `Y_POWERSTATE_FROM_EXT` corresponding to the current power source for module functions that require lots of current

On failure, throws an exception or returns `Y_POWERSTATE_INVALID`.

dualpower→set_logicalName() YDualPower
dualpower→setLogicalName()YDualPower
set_logicalName

Changes the logical name of the power control.

```
YDualPower 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 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()YDualPower
set_powerControl

Changes the selected power source for module functions that require lots of current.

YDualPower **target set_powerControl newval**

Parameters :

newval a value among `Y_POWERCONTROL_AUTO`, `Y_POWERCONTROL_FROM_USB`, `Y_POWERCONTROL_FROM_EXT` and `Y_POWERCONTROL_OFF` corresponding to the selected power source for module functions that require lots of current

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.17. Files function interface

The filesystem interface makes it possible to store files on some devices, for instance to design a custom web UI (for networked devices) or to add fonts (on display devices).

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_files.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YFiles = yoctolib.YFiles;
php     require_once('yocto_files.php');
cpp     #include "yocto_files.h"
m       #import "yocto_files.h"
pas     uses yocto_files;
vb      yocto_files.vb
cs      yocto_files.cs
java    import com.yoctopuce.YoctoAPI.YFiles;
py      from yocto_files import *
```

Global functions

yFindFiles(**func**)

Retrieves a filesystem for a given identifier.

yFirstFiles()

Starts the enumeration of filesystems currently accessible.

YFiles methods

files→describe()

Returns a short text that describes unambiguously the instance of the filesystem in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

files→download(**pathname**)

Downloads the requested file and returns a binary buffer with its content.

files→download_async(**pathname**, **callback**, **context**)

Downloads the requested file and returns a binary buffer with its content.

files→format_fs()

Reinitialize the filesystem to its clean, unfragmented, empty state.

files→get_advertisedValue()

Returns the current value of the filesystem (no more than 6 characters).

files→get_errorMessage()

Returns the error message of the latest error with the filesystem.

files→get_errorType()

Returns the numerical error code of the latest error with the filesystem.

files→get_filesCount()

Returns the number of files currently loaded in the filesystem.

files→get_freeSpace()

Returns the free space for uploading new files to the filesystem, in bytes.

files→get_friendlyName()

Returns a global identifier of the filesystem in the format `MODULE_NAME.FUNCTION_NAME`.

files→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

files→get_functionId()

Returns the hardware identifier of the filesystem, without reference to the module.

files→get_hardwareId()

Returns the unique hardware identifier of the filesystem in the form `SERIAL.FUNCTIONID`.

files→get_list(**pattern**)

Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.

files→get_logicalName()

Returns the logical name of the filesystem.

files→get_module()

Gets the `YModule` object for the device on which the function is located.

files→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

files→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

files→isOnline()

Checks if the filesystem is currently reachable, without raising any error.

files→isOnline_async(**callback**, **context**)

Checks if the filesystem is currently reachable, without raising any error (asynchronous version).

files→load(**msValidity**)

Preloads the filesystem cache with a specified validity duration.

files→load_async(**msValidity**, **callback**, **context**)

Preloads the filesystem cache with a specified validity duration (asynchronous version).

files→nextFiles()

Continues the enumeration of filesystems started using `yFirstFiles()`.

files→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

files→remove(**pathname**)

Deletes a file, given by its full path name, from the filesystem.

files→set_logicalName(**newval**)

Changes the logical name of the filesystem.

files→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

files→upload(**pathname**, **content**)

Uploads a file to the filesystem, to the specified full path name.

files→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

files→download()YFiles download — YFiles

Downloads the requested file and returns a binary buffer with its content.

```
YFiles target download pathname
```

Parameters :

pathname path and name of the file to download

Returns :

a binary buffer with the file content

On failure, throws an exception or returns an empty content.

files→format_fs()YFiles format_fs YFiles

Reinitialize the filesystem to its clean, unfragmented, empty state.

YFiles **target** **format_fs**

All files previously uploaded are permanently lost.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→get_advertisedValue() YFiles
files→advertisedValue()YFiles get_advertisedValue

Returns the current value of the filesystem (no more than 6 characters).

YFiles **target get_advertisedValue**

Returns :
a string corresponding to the current value of the filesystem (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

files→get_filesCount() YFiles
files→filesCount()YFiles get_filesCount

Returns the number of files currently loaded in the filesystem.

YFiles **target** **get_filesCount**

Returns :
an integer corresponding to the number of files currently loaded in the filesystem

On failure, throws an exception or returns `Y_FILESCOUNT_INVALID`.

files→get_freeSpace() YFiles
files→freeSpace()YFiles get_freeSpace

Returns the free space for uploading new files to the filesystem, in bytes.

YFiles **target get_freeSpace**

Returns :
an integer corresponding to the free space for uploading new files to the filesystem, in bytes

On failure, throws an exception or returns Y_FREESPACE_INVALID.

files→get_list() YFiles
files→list()YFiles get_list

Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.

YFiles **target get_list pattern**

Parameters :

pattern an optional filter pattern, using star and question marks as wildcards. When an empty pattern is provided, all file records are returned.

Returns :

a list of `YFileRecord` objects, containing the file path and name, byte size and 32-bit CRC of the file content.

On failure, throws an exception or returns an empty list.

files→get_logicalName() YFiles
files→logicalName()YFiles get_logicalName

Returns the logical name of the filesystem.

YFiles **target get_logicalName**

Returns :
a string corresponding to the logical name of the filesystem.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

files→remove()YFiles remove YFiles

Deletes a file, given by its full path name, from the filesystem.

```
YFiles target remove pathname
```

Because of filesystem fragmentation, deleting a file may not always free up the whole space used by the file. However, rewriting a file with the same path name will always reuse any space not freed previously. If you need to ensure that no space is taken by previously deleted files, you can use `format_fs` to fully reinitialize the filesystem.

Parameters :

pathname path and name of the file to remove.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→set_logicalName() YFiles
files→setLogicalName()YFiles set_logicalName

Changes the logical name of the filesystem.

```
YFiles 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 filesystem.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→upload()YFiles upload YFiles

Uploads a file to the filesystem, to the specified full path name.

```
YFiles target upload pathname content
```

If a file already exists with the same path name, its content is overwritten.

Parameters :

pathname path and name of the new file to create

content binary buffer with the content to set

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.18. GenericSensor function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_genericsensor.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YGenericSensor = yoctolib.YGenericSensor;
php     require_once('yocto_genericsensor.php');
cpp     #include "yocto_genericsensor.h"
m       #import "yocto_genericsensor.h"
pas     uses yocto_genericsensor;
vb      yocto_genericsensor.vb
cs      yocto_genericsensor.cs
java    import com.yoctopuce.YoctoAPI.YGenericSensor;
py      from yocto_genericsensor import *
```

Global functions
yFindGenericSensor(**func**) 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_signalBias()

Returns the electric signal bias for zero shift adjustment.

genericsensor→get_signalRange()

Returns the electric signal range used by the sensor.

genericsensor→get_signalUnit()

Returns the measuring unit of the electrical signal used by the sensor.

genericsensor→get_signalValue()

Returns the measured value of the electrical signal used by the sensor.

genericsensor→get_unit()

Returns the measuring unit for the measure.

genericsensor→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

genericsensor→get_valueRange()

Returns the physical value range measured by the sensor.

genericsensor→isOnline()

Checks if the generic sensor is currently reachable, without raising any error.

genericsensor→isOnline_async(callback, context)

Checks if the generic sensor is currently reachable, without raising any error (asynchronous version).

genericsensor→load(msValidity)

Preloads the generic sensor cache with a specified validity duration.

genericsensor→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

genericsensor→load_async(msValidity, callback, context)

Preloads the generic sensor cache with a specified validity duration (asynchronous version).

genericsensor→nextGenericSensor()

Continues the enumeration of generic sensors started using `yFirstGenericSensor()`.

genericsensor→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

genericsensor→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

genericsensor→set_highestValue(newval)

Changes the recorded maximal value observed.

genericsensor→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

genericsensor→set_logicalName(newval)

Changes the logical name of the generic sensor.

genericsensor→set_lowestValue(newval)

Changes the recorded minimal value observed.

genericsensor→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

genericsensor→set_resolution(newval)

Changes the resolution of the measured physical values.

genericsensor→set_signalBias(newval)

Changes the electric signal bias for zero shift adjustment.

genericsensor→set_signalRange(newval)

Changes the electric signal range used by the sensor.

genericsensor→set_unit(newval)

Changes the measuring unit for the measured value.

genericsensor→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

genericsensor→set_valueRange(newval)

Changes the physical value range measured by the sensor.

genericsensor→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

genericsensor→zeroAdjust()

Adjusts the signal bias so that the current signal value is need precisely as zero.

genericsensor→calibrateFromPoints() YGenericSensor
YGenericSensor calibrateFromPoints

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

YGenericSensor **target calibrateFromPoints rawValues refValues**

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→get_advertisedValue() YGenericSensor
genericsensor→advertisedValue()YGenericSensor
get_advertisedValue

Returns the current value of the generic sensor (no more than 6 characters).

YGenericSensor **target get_advertisedValue**

Returns :
a string corresponding to the current value of the generic sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

genericsensor→get_currentRawValue() YGenericSensor genericsensor→currentRawValue()YGenericSensor get_currentRawValue

Returns the uncalibrated, unrounded raw value returned by the sensor.

YGenericSensor **target** **get_currentRawValue**

Returns :
a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

genericsensor→get_currentValue() YGenericSensor
genericsensor→currentValue()YGenericSensor
get_currentValue

Returns the current measured value.

YGenericSensor **target get_currentValue**

Returns :
a floating point number corresponding to the current measured value

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

genericsensor→get_highestValue() YGenericSensor genericsensor→highestValue()YGenericSensor get_highestValue

Returns the maximal value observed for the measure since the device was started.

YGenericSensor **target** **get_highestValue**

Returns :
a floating point number corresponding to the maximal value observed for the measure since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

genericsensor→get_logFrequency() YGenericSensor
genericsensor→logFrequency()YGenericSensor
get_logFrequency

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

YGenericSensor **target get_logFrequency**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

genericsensor→get_logicalName() YGenericSensor genericsensor→logicalName()YGenericSensor get_logicalName

Returns the logical name of the generic sensor.

YGenericSensor **target** **get_logicalName**

Returns :
a string corresponding to the logical name of the generic sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

genericsensor→get_lowestValue() YGenericSensor genericsensor→lowestValue()YGenericSensor get_lowestValue

Returns the minimal value observed for the measure since the device was started.

YGenericSensor **target get_lowestValue**

Returns :
a floating point number corresponding to the minimal value observed for the measure since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

genericsensor→get_recordedData() YGenericSensor
genericsensor→recordedData()YGenericSensor
get_recordedData

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YGenericSensor **target get_recordedData startTime endTime**

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

genericsensor→get_reportFrequency() YGenericSensor genericsensor→reportFrequency()YGenericSensor get_reportFrequency

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

YGenericSensor **target get_reportFrequency**

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

genericsensor→get_resolution() YGenericSensor genericsensor→resolution()YGenericSensor get_resolution

Returns the resolution of the measured values.

```
YGenericSensor target get_resolution
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

genericsensor→get_signalBias() YGenericSensor genericsensor→signalBias()YGenericSensor get_signalBias

Returns the electric signal bias for zero shift adjustment.

YGenericSensor **target get_signalBias**

A positive bias means that the signal is over-reporting the measure, while a negative bias means that the signal is underreporting the measure.

Returns :
a floating point number corresponding to the electric signal bias for zero shift adjustment

On failure, throws an exception or returns `Y_SIGNALBIAS_INVALID`.

genericsensor→get_signalRange() YGenericSensor genericsensor→signalRange()YGenericSensor get_signalRange

Returns the electric signal range used by the sensor.

YGenericSensor **target** **get_signalRange**

Returns :
a string corresponding to the electric signal range used by the sensor

On failure, throws an exception or returns `Y_SIGNALRANGE_INVALID`.

genericsensor→get_signalUnit() YGenericSensor
genericsensor→signalUnit()YGenericSensor
get_signalUnit

Returns the measuring unit of the electrical signal used by the sensor.

YGenericSensor **target get_signalUnit**

Returns :
a string corresponding to the measuring unit of the electrical signal used by the sensor

On failure, throws an exception or returns `Y_SIGNALUNIT_INVALID`.

genericsensor→get_signalValue() YGenericSensor genericsensor→signalValue()YGenericSensor get_signalValue

Returns the measured value of the electrical signal used by the sensor.

YGenericSensor **target** **get_signalValue**

Returns :
a floating point number corresponding to the measured value of the electrical signal used by the sensor

On failure, throws an exception or returns `Y_SIGNALVALUE_INVALID`.

genericsensor→get_unit() YGenericSensor
genericsensor→unit()YGenericSensor get_unit

Returns the measuring unit for the measure.

```
YGenericSensor target get_unit
```

Returns :
a string corresponding to the measuring unit for the measure

On failure, throws an exception or returns `Y_UNIT_INVALID`.

genericsensor→get_valueRange() YGenericSensor genericsensor→valueRange()YGenericSensor get_valueRange

Returns the physical value range measured by the sensor.

YGenericSensor **target** **get_valueRange**

Returns :
a string corresponding to the physical value range measured by the sensor

On failure, throws an exception or returns `Y_VALUERANGE_INVALID`.

genericsensor→loadCalibrationPoints() YGenericSensor
YGenericSensor loadCalibrationPoints

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

YGenericSensor **target loadCalibrationPoints rawValues refValues**

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_highestValue() YGenericSensor genericsensor→setHighestValue()YGenericSensor set_highestValue

Changes the recorded maximal value observed.

```
YGenericSensor target set_highestValue newval
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_logFrequency() YGenericSensor
genericsensor→setLogFrequency()YGenericSensor
set_logFrequency

Changes the datalogger recording frequency for this function.

```
YGenericSensor target set_logFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_logicalName() YGenericSensor genericsensor→setLogicalName()YGenericSensor set_logicalName

Changes the logical name of the generic sensor.

```
YGenericSensor 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 generic sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_lowestValue() YGenericSensor genericsensor→setLowestValue()YGenericSensor set_lowestValue

Changes the recorded minimal value observed.

YGenericSensor **target set_lowestValue newval**

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_reportFrequency() YGenericSensor
genericsensor→setReportFrequency()
YGenericSensor set_reportFrequency

Changes the timed value notification frequency for this function.

```
YGenericSensor target set_reportFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_resolution() YGenericSensor
genericsensor→setResolution()YGenericSensor
set_resolution

Changes the resolution of the measured physical values.

```
YGenericSensor target set_resolution newval
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_signalBias() YGenericSensor genericsensor→setSignalBias()YGenericSensor set_signalBias

Changes the electric signal bias for zero shift adjustment.

```
YGenericSensor target set_signalBias newval
```

If your electric signal reads positif when it should be zero, setup a positive signalBias of the same value to fix the zero shift.

Parameters :

newval a floating point number corresponding to the electric signal bias for zero shift adjustment

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_signalRange() YGenericSensor
genericsensor→setSignalRange()YGenericSensor
set_signalRange

Changes the electric signal range used by the sensor.

YGenericSensor **target set_signalRange newval**

Parameters :

newval a string corresponding to the electric signal range used by the sensor

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_unit() YGenericSensor genericsensor→setUnit()YGenericSensor set_unit

Changes the measuring unit for the measured value.

```
YGenericSensor target set_unit newval
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the measuring unit for the measured value

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_valueRange() YGenericSensor genericsensor→setValueRange()YGenericSensor set_valueRange

Changes the physical value range measured by the sensor.

```
YGenericSensor target set_valueRange newval
```

As a side effect, the range modification may automatically modify the display resolution.

Parameters :

newval a string corresponding to the physical value range measured by the sensor

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→zeroAdjust()YGenericSensor zeroAdjust YGenericSensor

Adjusts the signal bias so that the current signal value is need precisely as zero.

YGenericSensor **target** **zeroAdjust**

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.19. Gyroscope function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
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');
cpp     #include "yocto_gyro.h"
m       #import "yocto_gyro.h"
pas     uses yocto_gyro;
vb      yocto_gyro.vb
cs      yocto_gyro.cs
java    import com.yoctopuce.YoctoAPI.YGyro;
py      from yocto_gyro import *
```

Global functions
yFindGyro(**func**) Retrieves a gyroscope for a given identifier.
yFirstGyro() Starts the enumeration of gyroscopes currently accessible.
YGyro methods
gyro→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
gyro→describe() Returns a short text that describes unambiguously the instance of the gyroscope in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
gyro→get_advertisedValue() Returns the current value of the gyroscope (no more than 6 characters).
gyro→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in degrees per second, as a floating point number.
gyro→get_currentValue() Returns the current value of the angular velocity, in degrees per second, as a floating point number.
gyro→get_errorMessage() Returns the error message of the latest error with the gyroscope.
gyro→get_errorType() Returns the numerical error code of the latest error with the gyroscope.
gyro→get_friendlyName() Returns a global identifier of the gyroscope in the format `MODULE_NAME.FUNCTION_NAME`.
gyro→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
gyro→get_functionId() Returns the hardware identifier of the gyroscope, without reference to the module.
gyro→get_hardwareId()

Returns the unique hardware identifier of the gyroscope in the form SERIAL.FUNCTIONID.

gyro→get_heading()

Returns the estimated heading angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_highestValue()

Returns the maximal value observed for the angular velocity since the device was started.

gyro→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

gyro→get_logicalName()

Returns the logical name of the gyroscope.

gyro→get_lowestValue()

Returns the minimal value observed for the angular velocity since the device was started.

gyro→get_module()

Gets the YModule object for the device on which the function is located.

gyro→get_module_async(**callback**, **context**)

Gets the YModule object for the device on which the function is located (asynchronous version).

gyro→get_pitch()

Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionW()

Returns the w component (real part) of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionX()

Returns the x component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionY()

Returns the y component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionZ()

Returns the x component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

gyro→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

gyro→get_resolution()

Returns the resolution of the measured values.

gyro→get_roll()

Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_unit()

Returns the measuring unit for the angular velocity.

gyro→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

gyro→get_xValue()

Returns the angular velocity around the X axis of the device, as a floating point number.

gyro→get_yValue()
Returns the angular velocity around the Y axis of the device, as a floating point number.

gyro→get_zValue()
Returns the angular velocity around the Z axis of the device, as a floating point number.

gyro→isOnline()
Checks if the gyroscope is currently reachable, without raising any error.

gyro→isOnline_async(**callback**, **context**)
Checks if the gyroscope is currently reachable, without raising any error (asynchronous version).

gyro→load(**msValidity**)
Preloads the gyroscope cache with a specified validity duration.

gyro→loadCalibrationPoints(**rawValues**, **refValues**)
Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

gyro→load_async(**msValidity**, **callback**, **context**)
Preloads the gyroscope cache with a specified validity duration (asynchronous version).

gyro→nextGyro()
Continues the enumeration of gyroscopes started using `yFirstGyro()`.

gyro→registerAnglesCallback(**callback**)
Registers a callback function that will be invoked each time that the estimated device orientation has changed.

gyro→registerQuaternionCallback(**callback**)
Registers a callback function that will be invoked each time that the estimated device orientation has changed.

gyro→registerTimedReportCallback(**callback**)
Registers the callback function that is invoked on every periodic timed notification.

gyro→registerValueCallback(**callback**)
Registers the callback function that is invoked on every change of advertised value.

gyro→set_highestValue(**newval**)
Changes the recorded maximal value observed.

gyro→set_logFrequency(**newval**)
Changes the datalogger recording frequency for this function.

gyro→set_logicalName(**newval**)
Changes the logical name of the gyroscope.

gyro→set_lowestValue(**newval**)
Changes the recorded minimal value observed.

gyro→set_reportFrequency(**newval**)
Changes the timed value notification frequency for this function.

gyro→set_resolution(**newval**)
Changes the resolution of the measured physical values.

gyro→set_userData(**data**)
Stores a user context provided as argument in the userData attribute of the function.

gyro→wait_async(**callback**, **context**)
Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

gyro→calibrateFromPoints()YGyro calibrateFromPoints YGyro

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

YGyro **target calibrateFromPoints rawValues refValues**

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→get_advertisedValue() YGyro
gyro→advertisedValue()YGyro get_advertisedValue

Returns the current value of the gyroscope (no more than 6 characters).

YGyro **target get_advertisedValue**

Returns :

a string corresponding to the current value of the gyroscope (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

gyro→get_currentRawValue() YGyro
gyro→currentRawValue()YGyro
get_currentRawValue

Returns the uncalibrated, unrounded raw value returned by the sensor, in degrees per second, as a floating point number.

YGyro **target get_currentRawValue**

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in degrees per second, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

gyro→get_currentValue() YGyro
gyro→currentValue()YGyro get_currentValue

Returns the current value of the angular velocity, in degrees per second, as a floating point number.

YGyro **target get_currentValue**

Returns :

a floating point number corresponding to the current value of the angular velocity, in degrees per second, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

gyro→get_highestValue() YGyro gyro→highestValue()YGyro get_highestValue

Returns the maximal value observed for the angular velocity since the device was started.

YGyro **target** **get_highestValue**

Returns :

a floating point number corresponding to the maximal value observed for the angular velocity since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

gyro→get_logFrequency() YGyro
gyro→logFrequency()YGyro get_logFrequency

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

YGyro **target get_logFrequency**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

gyro→get_logicalName() YGyro
gyro→logicalName()YGyro get_logicalName

Returns the logical name of the gyroscope.

YGyro **target** **get_logicalName**

Returns :
a string corresponding to the logical name of the gyroscope.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

gyro→get_lowestValue() YGyro
gyro→lowestValue()YGyro get_lowestValue

Returns the minimal value observed for the angular velocity since the device was started.

YGyro **target get_lowestValue**

Returns :

a floating point number corresponding to the minimal value observed for the angular velocity since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

gyro→get_recordedData() YGyro
gyro→recordedData()YGyro get_recordedData

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YGyro **target get_recordedData startTime endTime**

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

gyro→get_reportFrequency() YGyro
gyro→reportFrequency()YGyro get_reportFrequency

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
YGyro target get_reportFrequency
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

gyro→get_resolution() YGyro gyro→resolution()YGyro get_resolution

Returns the resolution of the measured values.

```
YGyro target get_resolution
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

gyro→get_unit() YGyro
gyro→unit()YGyro get_unit

Returns the measuring unit for the angular velocity.

YGyro **target get_unit**

Returns :
a string corresponding to the measuring unit for the angular velocity

On failure, throws an exception or returns `Y_UNIT_INVALID`.

gyro→loadCalibrationPoints()YGyro loadCalibrationPoints YGyro

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

YGyro **target loadCalibrationPoints rawValues refValues**

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_highestValue() YGyro
gyro→setHighestValue()YGyro set_highestValue

Changes the recorded maximal value observed.

YGyro **target set_highestValue newval**

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_logFrequency() YGyro
gyro→setLogFrequency()YGyro set_logFrequency

Changes the datalogger recording frequency for this function.

YGyro **target set_logFrequency newval**

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_logicalName() YGyro
gyro→setLogicalName()YGyro set_logicalName

Changes the logical name of the gyroscope.

YGyro **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 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()YGyro set_lowestValue

Changes the recorded minimal value observed.

```
YGyro target set_lowestValue newval
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_reportFrequency() YGyro
gyro→setReportFrequency()YGyro
set_reportFrequency

Changes the timed value notification frequency for this function.

```
YGyro target set_reportFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_resolution() YGyro gyro→setResolution()YGyro set_resolution

Changes the resolution of the measured physical values.

```
YGyro target set_resolution newval
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.20. Yocto-hub port interface

YHubPort objects provide control over the power supply for every YoctoHub port and provide information about the device connected to it. The logical name of a YHubPort is always automatically set to the unique serial number of the Yoctopuce device connected to it.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_hubport.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YHubPort = yoctolib.YHubPort;
php     require_once('yocto_hubport.php');
cpp     #include "yocto_hubport.h"
m       #import "yocto_hubport.h"
pas     uses yocto_hubport;
vb      yocto_hubport.vb
cs      yocto_hubport.cs
java    import com.yoctopuce.YoctoAPI.YHubPort;
py      from yocto_hubport import *
```

Global functions

yFindHubPort(**func**)

Retrieves a Yocto-hub port for a given identifier.

yFirstHubPort()

Starts the enumeration of Yocto-hub ports currently accessible.

`YHubPort` methods

hubport→describe()

Returns a short text that describes unambiguously the instance of the Yocto-hub port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

hubport→get_advertisedValue()

Returns the current value of the Yocto-hub port (no more than 6 characters).

hubport→get_baudRate()

Returns the current baud rate used by this Yocto-hub port, in kbps.

hubport→get_enabled()

Returns true if the Yocto-hub port is powered, false otherwise.

hubport→get_errorMessage()

Returns the error message of the latest error with the Yocto-hub port.

hubport→get_errorType()

Returns the numerical error code of the latest error with the Yocto-hub port.

hubport→get_friendlyName()

Returns a global identifier of the Yocto-hub port in the format `MODULE_NAME.FUNCTION_NAME`.

hubport→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

hubport→get_functionId()

Returns the hardware identifier of the Yocto-hub port, without reference to the module.

hubport→get_hardwareId()

Returns the unique hardware identifier of the Yocto-hub port in the form `SERIAL.FUNCTIONID`.

hubport→get_logicalName()

Returns the logical name of the Yocto-hub port.

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.

hubport→get_advertisedValue() YHubPort
hubport→advertisedValue()YHubPort
get_advertisedValue

Returns the current value of the Yocto-hub port (no more than 6 characters).

```
YHubPort target get_advertisedValue
```

Returns :
a string corresponding to the current value of the Yocto-hub port (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

hubport→get_baudRate() YHubPort hubport→baudRate()YHubPort get_baudRate

Returns the current baud rate used by this Yocto-hub port, in kbps.

```
YHubPort target get_baudRate
```

The default value is 1000 kbps, but a slower rate may be used if communication problems are encountered.

Returns :

an integer corresponding to the current baud rate used by this Yocto-hub port, in kbps

On failure, throws an exception or returns `Y_BAUDRATE_INVALID`.

hubport→get_enabled() YHubPort
hubport→enabled()YHubPort get_enabled

Returns true if the Yocto-hub port is powered, false otherwise.

YHubPort **target** **get_enabled**

Returns :

either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to true if the Yocto-hub port is powered, false otherwise

On failure, throws an exception or returns Y_ENABLED_INVALID.

hubport→get_logicalName() YHubPort hubport→logicalName()YHubPort get_logicalName

Returns the logical name of the Yocto-hub port.

YHubPort **target** **get_logicalName**

Returns :

a string corresponding to the logical name of the Yocto-hub port.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

hubport→get_portState() YHubPort
hubport→portState()YHubPort get_portState

Returns the current state of the Yocto-hub port.

```
YHubPort target get_portState
```

Returns :

a value among `Y_PORTSTATE_OFF`, `Y_PORTSTATE_OVRLD`, `Y_PORTSTATE_ON`, `Y_PORTSTATE_RUN` and `Y_PORTSTATE_PROG` corresponding to the current state of the Yocto-hub port

On failure, throws an exception or returns `Y_PORTSTATE_INVALID`.

hubport→set_enabled() YHubPort
hubport→setEnabled()YHubPort set_enabled

Changes the activation of the Yocto-hub port.

```
YHubPort target set_enabled newval
```

If the port is enabled, the connected module is powered. Otherwise, port power is shut down.

Parameters :

newval either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to the activation of the Yocto-hub port

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

hubport→set_logicalName() YHubPort
hubport→setLogicalName()YHubPort
set_logicalName

Changes the logical name of the Yocto-hub port.

YHubPort **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 Yocto-hub port.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.21. Humidity function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_humidity.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YHumidity = yoctolib.YHumidity;
php     require_once('yocto_humidity.php');
cpp     #include "yocto_humidity.h"
m       #import "yocto_humidity.h"
pas     uses yocto_humidity;
vb      yocto_humidity.vb
cs      yocto_humidity.cs
java    import com.yoctopuce.YoctoAPI.YHumidity;
py      from yocto_humidity import *
```

Global functions
yFindHumidity(**func**) Retrieves a humidity sensor for a given identifier.
yFirstHumidity() Starts the enumeration of humidity sensors currently accessible.
`YHumidity` methods
humidity→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
humidity→describe() Returns a short text that describes unambiguously the instance of the humidity sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
humidity→get_advertisedValue() Returns the current value of the humidity sensor (no more than 6 characters).
humidity→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in %RH, as a floating point number.
humidity→get_currentValue() Returns the current value of the humidity, in %RH, as a floating point number.
humidity→get_errorMessage() Returns the error message of the latest error with the humidity sensor.
humidity→get_errorType() Returns the numerical error code of the latest error with the humidity sensor.
humidity→get_friendlyName() Returns a global identifier of the humidity sensor in the format `MODULE_NAME.FUNCTION_NAME`.
humidity→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
humidity→get_functionId() Returns the hardware identifier of the humidity sensor, without reference to the module.
humidity→get_hardwareId() Returns the unique hardware identifier of the humidity sensor in the form `SERIAL.FUNCTIONID`.

humidity→get_highestValue()

Returns the maximal value observed for the humidity since the device was started.

humidity→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

humidity→get_logicalName()

Returns the logical name of the humidity sensor.

humidity→get_lowestValue()

Returns the minimal value observed for the humidity since the device was started.

humidity→get_module()

Gets the `YModule` object for the device on which the function is located.

humidity→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

humidity→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

humidity→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

humidity→get_resolution()

Returns the resolution of the measured values.

humidity→get_unit()

Returns the measuring unit for the humidity.

humidity→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

humidity→isOnline()

Checks if the humidity sensor is currently reachable, without raising any error.

humidity→isOnline_async(**callback**, **context**)

Checks if the humidity sensor is currently reachable, without raising any error (asynchronous version).

humidity→load(**msValidity**)

Preloads the humidity sensor cache with a specified validity duration.

humidity→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

humidity→load_async(**msValidity**, **callback**, **context**)

Preloads the humidity sensor cache with a specified validity duration (asynchronous version).

humidity→nextHumidity()

Continues the enumeration of humidity sensors started using `yFirstHumidity()`.

humidity→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

humidity→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

humidity→set_highestValue(**newval**)

Changes the recorded maximal value observed.

humidity→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

humidity→set_logicalName(**newval**)

Changes the logical name of the humidity sensor.

humidity→set_lowestValue(newval)

Changes the recorded minimal value observed.

humidity→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

humidity→set_resolution(newval)

Changes the resolution of the measured physical values.

humidity→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

humidity→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

humidity→calibrateFromPoints()YHumidity calibrateFromPoints — YHumidity

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

YHumidity **target calibrateFromPoints rawValues refValues**

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→get_advertisedValue() YHumidity
humidity→advertisedValue()YHumidity
get_advertisedValue

Returns the current value of the humidity sensor (no more than 6 characters).

YHumidity **target** **get_advertisedValue**

Returns :
a string corresponding to the current value of the humidity sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

humidity→get_currentRawValue() YHumidity
humidity→currentRawValue()YHumidity
get_currentRawValue

Returns the uncalibrated, unrounded raw value returned by the sensor, in %RH, as a floating point number.

YHumidity **target get_currentRawValue**

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in %RH, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

humidity→get_currentValue() YHumidity
humidity→currentValue()YHumidity get_currentValue

Returns the current value of the humidity, in %RH, as a floating point number.

```
YHumidity target get_currentValue
```

Returns :

a floating point number corresponding to the current value of the humidity, in %RH, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

humidity→get_highestValue() YHumidity
humidity→highestValue()YHumidity
get_highestValue

Returns the maximal value observed for the humidity since the device was started.

```
YHumidity target get_highestValue
```

Returns :

a floating point number corresponding to the maximal value observed for the humidity since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

humidity→get_logFrequency() YHumidity
humidity→logFrequency()YHumidity
get_logFrequency

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

YHumidity **target** **get_logFrequency**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

humidity→get_logicalName() YHumidity
humidity→logicalName()YHumidity get_logicalName

Returns the logical name of the humidity sensor.

YHumidity **target get_logicalName**

Returns :
a string corresponding to the logical name of the humidity sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

humidity→get_lowestValue() YHumidity
humidity→lowestValue()YHumidity get_lowestValue

Returns the minimal value observed for the humidity since the device was started.

YHumidity **target** **get_lowestValue**

Returns :

a floating point number corresponding to the minimal value observed for the humidity since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

humidity→get_recordedData() YHumidity
humidity→recordedData()YHumidity
get_recordedData

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YHumidity **target get_recordedData startTime endTime**

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

humidity→get_reportFrequency() YHumidity
humidity→reportFrequency()YHumidity
get_reportFrequency

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
YHumidity target get_reportFrequency
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

humidity→get_resolution() YHumidity
humidity→resolution()YHumidity get_resolution

Returns the resolution of the measured values.

```
YHumidity target get_resolution
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

humidity→get_unit() YHumidity
humidity→unit()YHumidity get_unit

Returns the measuring unit for the humidity.

```
YHumidity target get_unit
```

Returns :

a string corresponding to the measuring unit for the humidity

On failure, throws an exception or returns `Y_UNIT_INVALID`.

humidity→loadCalibrationPoints()YHumidity loadCalibrationPoints YHumidity

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

YHumidity **target loadCalibrationPoints rawValues refValues**

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_highestValue() YHumidity
humidity→setHighestValue()YHumidity
set_highestValue

Changes the recorded maximal value observed.

```
YHumidity target set_highestValue newval
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_logFrequency() YHumidity
humidity→setLogFrequency()YHumidity
set_logFrequency

Changes the datalogger recording frequency for this function.

```
YHumidity target set_logFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_logicalName() YHumidity
humidity→setLogicalName()YHumidity
set_logicalName

Changes the logical name of the humidity sensor.

```
YHumidity 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 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()YHumidity
set_lowestValue

Changes the recorded minimal value observed.

```
YHumidity target set_lowestValue newval
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_reportFrequency() YHumidity
humidity→setReportFrequency()YHumidity
set_reportFrequency

Changes the timed value notification frequency for this function.

```
YHumidity target set_reportFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

humidity→set_resolution() YHumidity humidity→setResolution()YHumidity set_resolution

Changes the resolution of the measured physical values.

```
YHumidity target set_resolution newval
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.22. Led function interface

Yoctopuce application programming interface allows you not only to drive the intensity of the led, but also to have it blink at various preset frequencies.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_led.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YLed = yoctolib.YLed;
php     require_once('yocto_led.php');
cpp     #include "yocto_led.h"
m       #import "yocto_led.h"
pas     uses yocto_led;
vb      yocto_led.vb
cs      yocto_led.cs
java    import com.yoctopuce.YoctoAPI.YLed;
py      from yocto_led import *
```

Global functions
yFindLed(**func**) 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.

led→get_advertisedValue() YLed
led→advertisedValue()YLed get_advertisedValue

Returns the current value of the led (no more than 6 characters).

```
YLed target get_advertisedValue
```

Returns :

a string corresponding to the current value of the led (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

led→get_blinking() YLed
led→blinking()YLed get_blinking

Returns the current led signaling mode.

```
YLed target get_blinking
```

Returns :

a value among `Y_BLINKING_STILL`, `Y_BLINKING_RELAX`, `Y_BLINKING_AWARE`, `Y_BLINKING_RUN`, `Y_BLINKING_CALL` and `Y_BLINKING_PANIC` corresponding to the current led signaling mode

On failure, throws an exception or returns `Y_BLINKING_INVALID`.

led→get_logicalName() YLed
led→logicalName()YLed get_logicalName

Returns the logical name of the led.

```
YLed target get_logicalName
```

Returns :

a string corresponding to the logical name of the led.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

led→get_luminosity() YLed
led→luminosity()YLed get_luminosity

Returns the current led intensity (in per cent).

```
YLed target get_luminosity
```

Returns :

an integer corresponding to the current led intensity (in per cent)

On failure, throws an exception or returns `Y_LUMINOSITY_INVALID`.

led→get_power() YLed led→power()YLed get_power

Returns the current led state.

```
YLed target get_power
```

Returns :

either `Y_POWER_OFF` or `Y_POWER_ON`, according to the current led state

On failure, throws an exception or returns `Y_POWER_INVALID`.

led→set_blinking() YLed
led→setBlinking()YLed set_blinking

Changes the current led signaling mode.

YLed **target** **set_blinking** **newval**

Parameters :

newval a value among `Y_BLINKING_STILL`, `Y_BLINKING_RELAX`, `Y_BLINKING_AWARE`, `Y_BLINKING_RUN`, `Y_BLINKING_CALL` and `Y_BLINKING_PANIC` corresponding to the current led signaling mode

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

led→set_logicalName() YLed led→setLogicalName()YLed set_logicalName

Changes the logical name of the led.

```
YLed 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 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()YLed set_luminosity

Changes the current led intensity (in per cent).

```
YLed target set_luminosity newval
```

Parameters :

newval an integer corresponding to the current led intensity (in per cent)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

led→set_power() YLed
led→setPower()YLed set_power

Changes the state of the led.

```
YLed target set_power newval
```

Parameters :

newval either Y_POWER_OFF or Y_POWER_ON, according to the state of the led

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.23. LightSensor function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_lightsensor.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YLightSensor = yoctolib.YLightSensor;
php     require_once('yocto_lightsensor.php');
cpp     #include "yocto_lightsensor.h"
m       #import "yocto_lightsensor.h"
pas     uses yocto_lightsensor;
vb      yocto_lightsensor.vb
cs      yocto_lightsensor.cs
java    import com.yoctopuce.YoctoAPI.YLightSensor;
py      from yocto_lightsensor import *
```

Global functions
yFindLightSensor(**func**) Retrieves a light sensor for a given identifier.
yFirstLightSensor() Starts the enumeration of light sensors currently accessible.
`YLightSensor` methods
lightsensor→calibrate(**calibratedVal**) Changes the sensor-specific calibration parameter so that the current value matches a desired target (linear scaling).
lightsensor→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
lightsensor→describe() Returns a short text that describes unambiguously the instance of the light sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
lightsensor→get_advertisedValue() Returns the current value of the light sensor (no more than 6 characters).
lightsensor→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in lux, as a floating point number.
lightsensor→get_currentValue() Returns the current value of the ambient light, in lux, as a floating point number.
lightsensor→get_errorMessage() Returns the error message of the latest error with the light sensor.
lightsensor→get_errorType() Returns the numerical error code of the latest error with the light sensor.
lightsensor→get_friendlyName() Returns a global identifier of the light sensor in the format `MODULE_NAME.FUNCTION_NAME`.
lightsensor→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
lightsensor→get_functionId()

Returns the hardware identifier of the light sensor, without reference to the module.

lightsensor→get_hardwareId()

Returns the unique hardware identifier of the light sensor in the form `SERIAL.FUNCTIONID`.

lightsensor→get_highestValue()

Returns the maximal value observed for the ambient light since the device was started.

lightsensor→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

lightsensor→get_logicalName()

Returns the logical name of the light sensor.

lightsensor→get_lowestValue()

Returns the minimal value observed for the ambient light since the device was started.

lightsensor→get_measureType()

Returns the type of light measure.

lightsensor→get_module()

Gets the `YModule` object for the device on which the function is located.

lightsensor→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

lightsensor→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

lightsensor→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

lightsensor→get_resolution()

Returns the resolution of the measured values.

lightsensor→get_unit()

Returns the measuring unit for the ambient light.

lightsensor→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

lightsensor→isOnline()

Checks if the light sensor is currently reachable, without raising any error.

lightsensor→isOnline_async(**callback**, **context**)

Checks if the light sensor is currently reachable, without raising any error (asynchronous version).

lightsensor→load(**msValidity**)

Preloads the light sensor cache with a specified validity duration.

lightsensor→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

lightsensor→load_async(**msValidity**, **callback**, **context**)

Preloads the light sensor cache with a specified validity duration (asynchronous version).

lightsensor→nextLightSensor()

Continues the enumeration of light sensors started using `yFirstLightSensor()`.

lightsensor→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

lightsensor→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

lightsensor→set_highestValue(**newval**)

Changes the recorded maximal value observed.

lightsensor→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

lightsensor→set_logicalName(newval)

Changes the logical name of the light sensor.

lightsensor→set_lowestValue(newval)

Changes the recorded minimal value observed.

lightsensor→set_measureType(newval)

Modify the light sensor type used in the device.

lightsensor→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

lightsensor→set_resolution(newval)

Changes the resolution of the measured physical values.

lightsensor→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

lightsensor→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

lightsensor→calibrate()YLightSensor calibrate YLightSensor

Changes the sensor-specific calibration parameter so that the current value matches a desired target (linear scaling).

```
YLightSensor target calibrate calibratedVal
```

Parameters :

calibratedVal the desired target value.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→calibrateFromPoints()YLightSensor calibrateFromPoints — YLightSensor

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

YLightSensor **target calibrateFromPoints rawValues refValues**

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→get_advertisedValue() YLightSensor
lightsensor→advertisedValue()YLightSensor
get_advertisedValue

Returns the current value of the light sensor (no more than 6 characters).

```
YLightSensor target get_advertisedValue
```

Returns :
a string corresponding to the current value of the light sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

lightsensor→get_currentRawValue() YLightSensor
lightsensor→currentRawValue()YLightSensor
get_currentRawValue

Returns the uncalibrated, unrounded raw value returned by the sensor, in lux, as a floating point number.

YLightSensor **target** **get_currentRawValue**

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in lux, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

lightsensor→get_currentValue() YLightSensor lightsensor→currentValue()YLightSensor get_currentValue

Returns the current value of the ambient light, in lux, as a floating point number.

YLightSensor **target** **get_currentValue**

Returns :

a floating point number corresponding to the current value of the ambient light, in lux, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

lightsensor→get_highestValue() YLightSensor
lightsensor→highestValue()YLightSensor
get_highestValue

Returns the maximal value observed for the ambient light since the device was started.

```
YLightSensor target get_highestValue
```

Returns :

a floating point number corresponding to the maximal value observed for the ambient light since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

lightsensor→get_logFrequency() YLightSensor
lightsensor→logFrequency()YLightSensor
get_logFrequency

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

YLightSensor **target** **get_logFrequency**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

lightsensor→get_logicalName() YLightSensor
lightsensor→logicalName()YLightSensor
get_logicalName

Returns the logical name of the light sensor.

```
YLightSensor target get_logicalName
```

Returns :
a string corresponding to the logical name of the light sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

lightsensor→get_lowestValue() YLightSensor
lightsensor→lowestValue()YLightSensor
get_lowestValue

Returns the minimal value observed for the ambient light since the device was started.

YLightSensor **target** **get_lowestValue**

Returns :

a floating point number corresponding to the minimal value observed for the ambient light since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

lightsensor→get_measureType() YLightSensor
lightsensor→measureType()YLightSensor
get_measureType

Returns the type of light measure.

YLightSensor **target** **get_measureType**

Returns :

a value among `Y_MEASURETYPE_HUMAN_EYE`, `Y_MEASURETYPE_WIDE_SPECTRUM`, `Y_MEASURETYPE_INFRARED`, `Y_MEASURETYPE_HIGH_RATE` and `Y_MEASURETYPE_HIGH_ENERGY` corresponding to the type of light measure

On failure, throws an exception or returns `Y_MEASURETYPE_INVALID`.

lightsensor→get_recordedData() YLightSensor
lightsensor→recordedData()YLightSensor
get_recordedData

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YLightSensor **target get_recordedData startTime endTime**

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

lightsensor→get_reportFrequency() YLightSensor
lightsensor→reportFrequency()YLightSensor
get_reportFrequency

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

YLightSensor **target** **get_reportFrequency**

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

lightsensor→get_resolution() YLightSensor
lightsensor→resolution()YLightSensor
get_resolution

Returns the resolution of the measured values.

```
YLightSensor target get_resolution
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

lightsensor→get_unit() YLightSensor
lightsensor→unit()YLightSensor get_unit

Returns the measuring unit for the ambient light.

YLightSensor **target** **get_unit**

Returns :
a string corresponding to the measuring unit for the ambient light

On failure, throws an exception or returns `Y_UNIT_INVALID`.

lightsensor→loadCalibrationPoints()YLightSensor YLightSensor loadCalibrationPoints

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

YLightSensor **target loadCalibrationPoints rawValues refValues**

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_highestValue() YLightSensor
lightsensor→setHighestValue()YLightSensor
set_highestValue

Changes the recorded maximal value observed.

```
YLightSensor target set_highestValue newval
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_logFrequency() YLightSensor lightsensor→setLogFrequency()YLightSensor set_logFrequency

Changes the datalogger recording frequency for this function.

```
YLightSensor target set_logFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_logicalName() YLightSensor
lightsensor→setLogicalName()YLightSensor
set_logicalName

Changes the logical name of the light sensor.

```
YLightSensor 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 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()YLightSensor
set_lowestValue

Changes the recorded minimal value observed.

```
YLightSensor target set_lowestValue newval
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_measureType() YLightSensor
lightsensor→setMeasureType()YLightSensor
set_measureType

Modify the light sensor type used in the device.

```
YLightSensor target set_measureType newval
```

The measure can either approximate the response of the human eye, focus on a specific light spectrum, depending on the capabilities of the light-sensitive cell. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a value among `Y_MEASURETYPE_HUMAN_EYE`, `Y_MEASURETYPE_WIDE_SPECTRUM`, `Y_MEASURETYPE_INFRARED`, `Y_MEASURETYPE_HIGH_RATE` and `Y_MEASURETYPE_HIGH_ENERGY`

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_reportFrequency() YLightSensor lightsensor→setReportFrequency()YLightSensor set_reportFrequency

Changes the timed value notification frequency for this function.

```
YLightSensor target set_reportFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

lightsensor→set_resolution() YLightSensor
lightsensor→setResolution()YLightSensor
set_resolution

Changes the resolution of the measured physical values.

```
YLightSensor target set_resolution newval
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.24. Magnetometer function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_magnetometer.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YMagnetometer = yoctolib.YMagnetometer;
php     require_once('yocto_magnetometer.php');
cpp     #include "yocto_magnetometer.h"
m       #import "yocto_magnetometer.h"
pas     uses yocto_magnetometer;
vb      yocto_magnetometer.vb
cs      yocto_magnetometer.cs
java    import com.yoctopuce.YoctoAPI.YMagnetometer;
py      from yocto_magnetometer import *
```

Global functions
yFindMagnetometer(**func**) Retrieves a magnetometer for a given identifier.
yFirstMagnetometer() Starts the enumeration of magnetometers currently accessible.
`YMagnetometer` methods
magnetometer→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
magnetometer→describe() Returns a short text that describes unambiguously the instance of the magnetometer in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
magnetometer→get_advertisedValue() Returns the current value of the magnetometer (no more than 6 characters).
magnetometer→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in mT, as a floating point number.
magnetometer→get_currentValue() Returns the current value of the magnetic field, in mT, as a floating point number.
magnetometer→get_errorMessage() Returns the error message of the latest error with the magnetometer.
magnetometer→get_errorType() Returns the numerical error code of the latest error with the magnetometer.
magnetometer→get_friendlyName() Returns a global identifier of the magnetometer in the format `MODULE_NAME.FUNCTION_NAME`.
magnetometer→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
magnetometer→get_functionId() Returns the hardware identifier of the magnetometer, without reference to the module.
magnetometer→get_hardwareId() Returns the unique hardware identifier of the magnetometer in the form `SERIAL.FUNCTIONID`.

magnetometer→get_highestValue()

Returns the maximal value observed for the magnetic field since the device was started.

magnetometer→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

magnetometer→get_logicalName()

Returns the logical name of the magnetometer.

magnetometer→get_lowestValue()

Returns the minimal value observed for the magnetic field since the device was started.

magnetometer→get_module()

Gets the `YModule` object for the device on which the function is located.

magnetometer→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

magnetometer→get_recordedData(startTime, endTime)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

magnetometer→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

magnetometer→get_resolution()

Returns the resolution of the measured values.

magnetometer→get_unit()

Returns the measuring unit for the magnetic field.

magnetometer→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

magnetometer→get_xValue()

Returns the X component of the magnetic field, as a floating point number.

magnetometer→get_yValue()

Returns the Y component of the magnetic field, as a floating point number.

magnetometer→get_zValue()

Returns the Z component of the magnetic field, as a floating point number.

magnetometer→isOnline()

Checks if the magnetometer is currently reachable, without raising any error.

magnetometer→isOnline_async(callback, context)

Checks if the magnetometer is currently reachable, without raising any error (asynchronous version).

magnetometer→load(msValidity)

Preloads the magnetometer cache with a specified validity duration.

magnetometer→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

magnetometer→load_async(msValidity, callback, context)

Preloads the magnetometer cache with a specified validity duration (asynchronous version).

magnetometer→nextMagnetometer()

Continues the enumeration of magnetometers started using `yFirstMagnetometer()`.

magnetometer→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

magnetometer→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

magnetometer→set_highestValue(newval)

Changes the recorded maximal value observed.

magnetometer→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

magnetometer→set_logicalName(newval)

Changes the logical name of the magnetometer.

magnetometer→set_lowestValue(newval)

Changes the recorded minimal value observed.

magnetometer→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

magnetometer→set_resolution(newval)

Changes the resolution of the measured physical values.

magnetometer→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

magnetometer→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

magnetometer→calibrateFromPoints() YMagnetometer
YMagnetometer calibrateFromPoints

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

YMagnetometer **target calibrateFromPoints rawValues refValues**

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→get_advertisedValue() YMagnetometer
magnetometer→advertisedValue()YMagnetometer get_advertisedValue

Returns the current value of the magnetometer (no more than 6 characters).

YMagnetometer **target get_advertisedValue**

Returns :
a string corresponding to the current value of the magnetometer (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

magnetometer→get_currentRawValue() YMagnetometer
magnetometer→currentRawValue()YMagnetometer
get_currentRawValue

Returns the uncalibrated, unrounded raw value returned by the sensor, in mT, as a floating point number.

YMagnetometer **target** **get_currentRawValue**

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in mT, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

magnetometer→get_currentValue() YMagnetometer
magnetometer→currentValue()YMagnetometer
get_currentValue

Returns the current value of the magnetic field, in mT, as a floating point number.

YMagnetometer **target** **get_currentValue**

Returns :
a floating point number corresponding to the current value of the magnetic field, in mT, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

magnetometer→get_highestValue() YMagnetometer
magnetometer→highestValue()YMagnetometer
get_highestValue

Returns the maximal value observed for the magnetic field since the device was started.

YMagnetometer **target** **get_highestValue**

Returns :
a floating point number corresponding to the maximal value observed for the magnetic field since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

magnetometer→get_logFrequency() YMagnetometer
magnetometer→logFrequency()YMagnetometer
get_logFrequency

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

YMagnetometer **target** **get_logFrequency**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

magnetometer→get_logicalName() YMagnetometer
magnetometer→logicalName()YMagnetometer
get_logicalName

Returns the logical name of the magnetometer.

YMagnetometer **target** **get_logicalName**

Returns :
a string corresponding to the logical name of the magnetometer.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

magnetometer→get_lowestValue() YMagnetometer
magnetometer→lowestValue()YMagnetometer
get_lowestValue

Returns the minimal value observed for the magnetic field since the device was started.

YMagnetometer **target get_lowestValue**

Returns :

a floating point number corresponding to the minimal value observed for the magnetic field since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

magnetometer→get_recordedData() YMagnetometer
magnetometer→recordedData()YMagnetometer
get_recordedData

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YMagnetometer **target get_recordedData startTime endTime**

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

magnetometer→get_reportFrequency() YMagnetometer magnetometer→reportFrequency()YMagnetometer get_reportFrequency

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

YMagnetometer **target** **get_reportFrequency**

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

magnetometer→get_resolution() YMagnetometer
magnetometer→resolution()YMagnetometer
get_resolution

Returns the resolution of the measured values.

YMagnetometer **target** **get_resolution**

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

magnetometer→get_unit() YMagnetometer
magnetometer→unit()YMagnetometer get_unit

Returns the measuring unit for the magnetic field.

YMagnetometer **target** **get_unit**

Returns :
a string corresponding to the measuring unit for the magnetic field

On failure, throws an exception or returns `Y_UNIT_INVALID`.

magnetometer→get_xValue() YMagnetometer
magnetometer→xValue()YMagnetometer get_xValue

Returns the X component of the magnetic field, as a floating point number.

YMagnetometer **target** **get_xValue**

Returns :
a floating point number corresponding to the X component of the magnetic field, as a floating point number

On failure, throws an exception or returns Y_XVALUE_INVALID.

magnetometer→get_yValue() YMagnetometer
magnetometer→yValue()YMagnetometer get_yValue

Returns the Y component of the magnetic field, as a floating point number.

YMagnetometer **target** **get_yValue**

Returns :
a floating point number corresponding to the Y component of the magnetic field, as a floating point number

On failure, throws an exception or returns `Y_YVALUE_INVALID`.

magnetometer→get_zValue() YMagnetometer
magnetometer→zValue()YMagnetometer get_zValue

Returns the Z component of the magnetic field, as a floating point number.

YMagnetometer **target** **get_zValue**

Returns :
a floating point number corresponding to the Z component of the magnetic field, as a floating point number

On failure, throws an exception or returns Y_ZVALUE_INVALID.

magnetometer→loadCalibrationPoints() YMagnetometer
YMagnetometer loadCalibrationPoints

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

YMagnetometer **target loadCalibrationPoints rawValues refValues**

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_highestValue() YMagnetometer
magnetometer→setHighestValue()YMagnetometer
set_highestValue

Changes the recorded maximal value observed.

YMagnetometer **target** **set_highestValue** **newval**

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_logFrequency() YMagnetometer magnetometer→setLogFrequency()YMagnetometer set_logFrequency

Changes the datalogger recording frequency for this function.

```
YMagnetometer target set_logFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_logicalName() YMagnetometer
magnetometer→setLogicalName()YMagnetometer
set_logicalName

Changes the logical name of the magnetometer.

```
YMagnetometer 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 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()YMagnetometer set_lowestValue

Changes the recorded minimal value observed.

YMagnetometer **target set_lowestValue newval**

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_reportFrequency() YMagnetometer
magnetometer→setReportFrequency()
YMagnetometer set_reportFrequency

Changes the timed value notification frequency for this function.

```
YMagnetometer target set_reportFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

magnetometer→set_resolution() YMagnetometer
magnetometer→setResolution()YMagnetometer
set_resolution

Changes the resolution of the measured physical values.

```
YMagnetometer target set_resolution newval
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.25. Measured value

YMeasure objects are used within the API to represent a value measured at a specified time. These objects are used in particular in conjunction with the YDataSet class.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_api.js'></script>
nodejs  var yoctolib = require('yoctolib');
        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).

3.26. Module control interface

This interface is identical for all Yoctopuce USB modules. It can be used to control the module global parameters, and to enumerate the functions provided by each module.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_api.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YAPI = yoctolib.YAPI;
        var YModule = yoctolib.YModule;
php     require_once('yocto_api.php');
cpp     #include "yocto_api.h"
m       #import "yocto_api.h"
pas     uses yocto_api;
vb      yocto_api.vb
cs      yocto_api.cs
java    import com.yoctopuce.YoctoAPI.YModule;
py      from yocto_api import *
```

Global functions
yFindModule(**func**) Allows you to find a module from its serial number or from its logical name.
yFirstModule() Starts the enumeration of modules currently accessible.
`YModule` methods
module→checkFirmware(**path**, **onlynew**) Test if the byn file is valid for this module.
module→describe() Returns a descriptive text that identifies the module.
module→download(**pathname**) Downloads the specified built-in file and returns a binary buffer with its content.
module→functionCount() Returns the number of functions (beside the "module" interface) available on the module.
module→functionId(**functionIndex**) Retrieves the hardware identifier of the *n*th function on the module.
module→functionName(**functionIndex**) Retrieves the logical name of the *n*th function on the module.
module→functionValue(**functionIndex**) Retrieves the advertised value of the *n*th function on the module.
module→get_allSettings() Returns all the setting of the module.
module→get_beacon() Returns the state of the localization beacon.
module→get_errorMessage() Returns the error message of the latest error with this module object.
module→get_errorType() Returns the numerical error code of the latest error with this module object.
module→get_firmwareRelease()

Returns the version of the firmware embedded in the module.

module→get_hardwareId()
Returns the unique hardware identifier of the module.

module→get_icon2d()
Returns the icon of the module.

module→get_lastLogs()
Returns a string with last logs of the module.

module→get_logicalName()
Returns the logical name of the module.

module→get_luminosity()
Returns the luminosity of the module informative leds (from 0 to 100).

module→get_persistentSettings()
Returns the current state of persistent module settings.

module→get_productId()
Returns the USB device identifier of the module.

module→get_productName()
Returns the commercial name of the module, as set by the factory.

module→get_productRelease()
Returns the hardware release version of the module.

module→get_rebootCountdown()
Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

module→get_serialNumber()
Returns the serial number of the module, as set by the factory.

module→get_upTime()
Returns the number of milliseconds spent since the module was powered on.

module→get_usbCurrent()
Returns the current consumed by the module on the USB bus, in milli-amps.

module→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

module→get_userVar()
Returns the value previously stored in this attribute.

module→isOnline()
Checks if the module is currently reachable, without raising any error.

module→isOnline_async(**callback**, **context**)
Checks if the module is currently reachable, without raising any error.

module→load(**msValidity**)
Preloads the module cache with a specified validity duration.

module→load_async(**msValidity**, **callback**, **context**)
Preloads the module cache with a specified validity duration (asynchronous version).

module→nextModule()
Continues the module enumeration started using `yFirstModule()`.

module→reboot(**secBeforeReboot**)
Schedules a simple module reboot after the given number of seconds.

module→registerLogCallback(**callback**)
Registers a device log callback function.

module→revertFromFlash()

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

module→saveToFlash()

Saves current settings in the nonvolatile memory of the module.

module→set_allSettings(**settings**)

Restore all the setting of the module.

module→set_beacon(**newval**)

Turns on or off the module localization beacon.

module→set_logicalName(**newval**)

Changes the logical name of the module.

module→set_luminosity(**newval**)

Changes the luminosity of the module informative leds.

module→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

module→set_userVar(**newval**)

Returns the value previously stored in this attribute.

module→triggerFirmwareUpdate(**secBeforeReboot**)

Schedules a module reboot into special firmware update mode.

module→updateFirmware(**path**)

Prepare a firmware upgrade of the module.

module→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

module→checkFirmware()YModule checkFirmware YModule

Test if the byn file is valid for this module.

YModule **target checkFirmware** **path onlynew**

This method is useful to test if the module need to be updated. It's possible to pass an directory instead of a file. In this case this method return the path of the most recent appropriate byn file. If the parameter onlynew is true the function will discard firmware that are older or equal to the installed firmware.

Parameters :

path	the path of a byn file or a directory that contain byn files
onlynew	return only files that are strictly newer

Returns :

: the path of the byn file to use or a empty string if no byn files match the requirement

On failure, throws an exception or returns a string that start with "error:".

module→download()YModule download YModule

Downloads the specified built-in file and returns a binary buffer with its content.

YModule **target** **download** **pathname**

Parameters :

pathname name of the new file to load

Returns :

a binary buffer with the file content

On failure, throws an exception or returns YAPI_INVALID_STRING.

module→get_allSettings() YModule
module→allSettings()YModule get_allSettings

Returns all the setting of the module.

YModule **target get_allSettings**

Useful to backup all the logical name and calibrations parameters of a connected module.

Returns :
a binary buffer with all settings.

On failure, throws an exception or returns `YAPI_INVALID_STRING`.

module→get_beacon() YModule
module→beacon()YModule get_beacon

Returns the state of the localization beacon.

YModule **target** **get_beacon**

Returns :

either `Y_BEACON_OFF` or `Y_BEACON_ON`, according to the state of the localization beacon

On failure, throws an exception or returns `Y_BEACON_INVALID`.

module→get_firmwareRelease() YModule
module→firmwareRelease()YModule
get_firmwareRelease

Returns the version of the firmware embedded in the module.

YModule **target get_firmwareRelease**

Returns :
a string corresponding to the version of the firmware embedded in the module

On failure, throws an exception or returns `Y_FIRMWARERELEASE_INVALID`.

module→get_icon2d() YModule
module→icon2d()YModule get_icon2d

Returns the icon of the module.

```
YModule target get_icon2d
```

The icon is a PNG image and does not exceeds 1536 bytes.

Returns :

a binary buffer with module icon, in png format. On failure, throws an exception or returns `YAPI_INVALID_STRING`.

module→get_lastLogs() YModule
module→lastLogs()YModule get_lastLogs

Returns a string with last logs of the module.

```
YModule target get_lastLogs
```

This method return only logs that are still in the module.

Returns :

a string with last logs of the module. On failure, throws an exception or returns `YAPI_INVALID_STRING`.

module→get_logicalName() YModule
module→logicalName()YModule get_logicalName

Returns the logical name of the module.

YModule **target** **get_logicalName**

Returns :
a string corresponding to the logical name of the module

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

module→get_luminosity() YModule
module→luminosity()YModule get_luminosity

Returns the luminosity of the module informative leds (from 0 to 100).

YModule **target get_luminosity**

Returns :
an integer corresponding to the luminosity of the module informative leds (from 0 to 100)

On failure, throws an exception or returns `Y_LUMINOSITY_INVALID`.

module→get_persistentSettings() YModule
module→persistentSettings()YModule
get_persistentSettings

Returns the current state of persistent module settings.

YModule **target** **get_persistentSettings**

Returns :

a value among `Y_PERSISTENTSETTINGS_LOADED`, `Y_PERSISTENTSETTINGS_SAVED` and `Y_PERSISTENTSETTINGS_MODIFIED` corresponding to the current state of persistent module settings

On failure, throws an exception or returns `Y_PERSISTENTSETTINGS_INVALID`.

module→get_productId() YModule
module→productId()YModule get_productId

Returns the USB device identifier of the module.

YModule **target get_productId**

Returns :
an integer corresponding to the USB device identifier of the module

On failure, throws an exception or returns `Y_PRODUCTID_INVALID`.

module→get_productName() YModule
module→productName()YModule get_productName

Returns the commercial name of the module, as set by the factory.

YModule **target** **get_productName**

Returns :
a string corresponding to the commercial name of the module, as set by the factory

On failure, throws an exception or returns `Y_PRODUCTNAME_INVALID`.

module→get_productRelease() YModule
module→productRelease()YModule
get_productRelease

Returns the hardware release version of the module.

YModule **target get_productRelease**

Returns :
an integer corresponding to the hardware release version of the module

On failure, throws an exception or returns `Y_PRODUCTRELEASE_INVALID`.

module→get_rebootCountdown() YModule
module→rebootCountdown()YModule
get_rebootCountdown

Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

YModule **target** **get_rebootCountdown**

Returns :

an integer corresponding to the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled

On failure, throws an exception or returns `Y_REBOOTCOUNTDOWN_INVALID`.

module→get_serialNumber() YModule
module→serialNumber()YModule get_serialNumber

Returns the serial number of the module, as set by the factory.

YModule **target get_serialNumber**

Returns :
a string corresponding to the serial number of the module, as set by the factory

On failure, throws an exception or returns `Y_SERIALNUMBER_INVALID`.

module→get_upTime() YModule
module→upTime()YModule get_upTime

Returns the number of milliseconds spent since the module was powered on.

YModule **target** **get_upTime**

Returns :

an integer corresponding to the number of milliseconds spent since the module was powered on

On failure, throws an exception or returns `Y_UPTIME_INVALID`.

module→get_usbCurrent() YModule
module→usbCurrent()YModule get_usbCurrent

Returns the current consumed by the module on the USB bus, in milli-amps.

YModule **target get_usbCurrent**

Returns :
an integer corresponding to the current consumed by the module on the USB bus, in milli-amps

On failure, throws an exception or returns `Y_USBCURRENT_INVALID`.

module→get_userVar() YModule
module→userVar()YModule get_userVar

Returns the value previously stored in this attribute.

```
YModule target get_userVar
```

On startup and after a device reboot, the value is always reset to zero.

Returns :
an integer corresponding to the value previously stored in this attribute

On failure, throws an exception or returns `Y_USERVAR_INVALID`.

module→reboot()YModule reboot YModule

Schedules a simple module reboot after the given number of seconds.

YModule **target reboot secBeforeReboot**

Parameters :

secBeforeReboot number of seconds before rebooting

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→revertFromFlash()YModule YModule revertFromFlash

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

YModule **target** **revertFromFlash**

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→saveToFlash()YModule saveToFlash YModule

Saves current settings in the nonvolatile memory of the module.

```
YModule target saveToFlash
```

Warning: the number of allowed save operations during a module life is limited (about 100000 cycles). Do not call this function within a loop.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_allSettings() YModule
module→setAllSettings()YModule set_allSettings

Restore all the setting of the module.

```
YModule target set_allSettings settings
```

Useful to restore all the logical name and calibrations parameters of a module from a backup.

Parameters :

settings a binary buffer with all settings.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_beacon() YModule
module→setBeacon()YModule set_beacon

Turns on or off the module localization beacon.

YModule **target** **set_beacon** **newval**

Parameters :

newval either Y_BEACON_OFF or Y_BEACON_ON

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_logicalName() YModule
module→setLogicalName()YModule set_logicalName

Changes the logical name of the module.

```
YModule 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 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()YModule set_luminosity

Changes the luminosity of the module informative leds.

YModule **target set_luminosity newval**

The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer corresponding to the luminosity of the module informative leds

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_userVar() YModule
module→setUserVar()YModule set_userVar

Returns the value previously stored in this attribute.

```
YModule target set_userVar newval
```

On startup and after a device reboot, the value is always reset to zero.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→triggerFirmwareUpdate()YModule triggerFirmwareUpdate

YModule

Schedules a module reboot into special firmware update mode.

YModule **target triggerFirmwareUpdate secBeforeReboot**

Parameters :

secBeforeReboot number of seconds before rebooting

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→updateFirmware()YModule updateFirmware YModule

Prepare a firmware upgrade of the module.

```
YModule target updateFirmware path
```

This method return a object `YFirmwareUpdate` which will handle the firmware upgrade process.

Parameters :

path the path of the byn file to use.

Returns :

: A object `YFirmwareUpdate`.

3.27. Motor function interface

Yoctopuce application programming interface allows you to drive the power sent to the motor to make it turn both ways, but also to drive accelerations and decelerations. The motor will then accelerate automatically: you will not have to monitor it. The API also allows to slow down the motor by shortening its terminals: the motor will then act as an electromagnetic brake.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_motor.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YMotor = yoctolib.YMotor;
php     require_once('yocto_motor.php');
cpp     #include "yocto_motor.h"
m       #import "yocto_motor.h"
pas     uses yocto_motor;
vb      yocto_motor.vb
cs      yocto_motor.cs
java    import com.yoctopuce.YoctoAPI.YMotor;
py      from yocto_motor import *
```

Global functions
yFindMotor(**func**) Retrieves a motor for a given identifier.
yFirstMotor() Starts the enumeration of motors currently accessible.
`YMotor` methods
motor→brakingForceMove(**targetPower**, **delay**) Changes progressively the braking force applied to the motor for a specific duration.
motor→describe() Returns a short text that describes unambiguously the instance of the motor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
motor→drivingForceMove(**targetPower**, **delay**) Changes progressively the power sent to the moteur for a specific duration.
motor→get_advertisedValue() Returns the current value of the motor (no more than 6 characters).
motor→get_brakingForce() Returns the braking force applied to the motor, as a percentage.
motor→get_cutOffVoltage() Returns the threshold voltage under which the controller automatically switches to error state and prevents further current draw.
motor→get_drivingForce() Returns the power sent to the motor, as a percentage between -100% and +100%.
motor→get_errorMessage() Returns the error message of the latest error with the motor.
motor→get_errorType() Returns the numerical error code of the latest error with the motor.
motor→get_failSafeTimeout() Returns the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process.

motor→get_frequency()
Returns the PWM frequency used to control the motor.

motor→get_friendlyName()
Returns a global identifier of the motor in the format `MODULE_NAME.FUNCTION_NAME`.

motor→get_functionDescriptor()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

motor→get_functionId()
Returns the hardware identifier of the motor, without reference to the module.

motor→get_hardwareId()
Returns the unique hardware identifier of the motor in the form `SERIAL.FUNCTIONID`.

motor→get_logicalName()
Returns the logical name of the motor.

motor→get_module()
Gets the `YModule` object for the device on which the function is located.

motor→get_module_async(**callback**, **context**)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

motor→get_motorStatus()
Return the controller state.

motor→get_overCurrentLimit()
Returns the current threshold (in mA) above which the controller automatically switches to error state.

motor→get_starterTime()
Returns the duration (in ms) during which the motor is driven at low frequency to help it start up.

motor→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

motor→isOnline()
Checks if the motor is currently reachable, without raising any error.

motor→isOnline_async(**callback**, **context**)
Checks if the motor is currently reachable, without raising any error (asynchronous version).

motor→keepALive()
Rearms the controller failsafe timer.

motor→load(**msValidity**)
Preloads the motor cache with a specified validity duration.

motor→load_async(**msValidity**, **callback**, **context**)
Preloads the motor cache with a specified validity duration (asynchronous version).

motor→nextMotor()
Continues the enumeration of motors started using `yFirstMotor()`.

motor→registerValueCallback(**callback**)
Registers the callback function that is invoked on every change of advertised value.

motor→resetStatus()
Reset the controller state to IDLE.

motor→set_brakingForce(**newval**)
Changes immediately the braking force applied to the motor (in percents).

motor→set_cutOffVoltage(**newval**)
Changes the threshold voltage under which the controller automatically switches to error state and prevents further current draw.

motor→set_drivingForce(newval)

Changes immediately the power sent to the motor.

motor→set_failSafeTimeout(newval)

Changes the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process.

motor→set_frequency(newval)

Changes the PWM frequency used to control the motor.

motor→set_logicalName(newval)

Changes the logical name of the motor.

motor→set_overCurrentLimit(newval)

Changes the current threshold (in mA) above which the controller automatically switches to error state.

motor→set_starterTime(newval)

Changes the duration (in ms) during which the motor is driven at low frequency to help it start up.

motor→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

motor→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

motor→brakingForceMove()YMotor brakingForceMove

YMotor

Changes progressively the braking force applied to the motor for a specific duration.

YMotor **target brakingForceMove targetPower delay**

Parameters :

targetPower	desired braking force, in percents
delay	duration (in ms) of the transition

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→drivingForceMove()YMotor drivingForceMove

YMotor

Changes progressively the power sent to the moteur for a specific duration.

YMotor **target drivingForceMove targetPower delay**

Parameters :

targetPower	desired motor power, in percents (between -100% and +100%)
delay	duration (in ms) of the transition

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→get_advertisedValue() YMotor
motor→advertisedValue()YMotor
get_advertisedValue

Returns the current value of the motor (no more than 6 characters).

YMotor **target** **get_advertisedValue**

Returns :
a string corresponding to the current value of the motor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

motor→get_brakingForce() YMotor
motor→brakingForce()YMotor get_brakingForce

Returns the braking force applied to the motor, as a percentage.

```
YMotor target get_brakingForce
```

The value 0 corresponds to no braking (free wheel).

Returns :
a floating point number corresponding to the braking force applied to the motor, as a percentage

On failure, throws an exception or returns `Y_BRAKINGFORCE_INVALID`.

motor→get_cutOffVoltage() YMotor motor→cutOffVoltage()YMotor get_cutOffVoltage

Returns the threshold voltage under which the controller automatically switches to error state and prevents further current draw.

```
YMotor target get_cutOffVoltage
```

This setting prevents damage to a battery that can occur when drawing current from an "empty" battery.

Returns :

a floating point number corresponding to the threshold voltage under which the controller automatically switches to error state and prevents further current draw

On failure, throws an exception or returns `Y_CUTOFFVOLTAGE_INVALID`.

motor→get_drivingForce() YMotor
motor→drivingForce()YMotor get_drivingForce

Returns the power sent to the motor, as a percentage between -100% and +100%.

YMotor **target** **get_drivingForce**

Returns :

a floating point number corresponding to the power sent to the motor, as a percentage between -100% and +100%

On failure, throws an exception or returns `Y_DRIVINGFORCE_INVALID`.

motor→get_failSafeTimeout() YMotor motor→failSafeTimeout()YMotor get_failSafeTimeout

Returns the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process.

```
YMotor target get_failSafeTimeout
```

When this delay has elapsed, the controller automatically stops the motor and switches to FAILSAFE error. Failsafe security is disabled when the value is zero.

Returns :

an integer corresponding to the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process

On failure, throws an exception or returns `Y_FAILSAFETIMEOUT_INVALID`.

motor→get_frequency() YMotor
motor→frequency()YMotor get_frequency

Returns the PWM frequency used to control the motor.

YMotor **target** **get_frequency**

Returns :

a floating point number corresponding to the PWM frequency used to control the motor

On failure, throws an exception or returns `Y_FREQUENCY_INVALID`.

motor→get_logicalName() YMotor
motor→logicalName()YMotor get_logicalName

Returns the logical name of the motor.

YMotor **target** **get_logicalName**

Returns :
a string corresponding to the logical name of the motor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

motor→get_motorStatus() YMotor motor→motorStatus()YMotor get_motorStatus

Return the controller state.

```
YMotor target get_motorStatus
```

Possible states are: IDLE when the motor is stopped/in free wheel, ready to start; FORWD when the controller is driving the motor forward; BACKWD when the controller is driving the motor backward; BRAKE when the controller is braking; LOVOLT when the controller has detected a low voltage condition; HICURR when the controller has detected an overcurrent condition; HIHEAT when the controller has detected an overheat condition; FAILSF when the controller switched on the failsafe security.

When an error condition occurred (LOVOLT, HICURR, HIHEAT, FAILSF), the controller status must be explicitly reset using the `resetStatus` function.

Returns :

a value among `Y_MOTORSTATUS_IDLE`, `Y_MOTORSTATUS_BRAKE`, `Y_MOTORSTATUS_FORWD`, `Y_MOTORSTATUS_BACKWD`, `Y_MOTORSTATUS_LOVOLT`, `Y_MOTORSTATUS_HICURR`, `Y_MOTORSTATUS_HIHEAT` and `Y_MOTORSTATUS_FAILSF`

On failure, throws an exception or returns `Y_MOTORSTATUS_INVALID`.

motor→get_overCurrentLimit() YMotor
motor→overCurrentLimit()YMotor
get_overCurrentLimit

Returns the current threshold (in mA) above which the controller automatically switches to error state.

```
YMotor target get_overCurrentLimit
```

A zero value means that there is no limit.

Returns :

an integer corresponding to the current threshold (in mA) above which the controller automatically switches to error state

On failure, throws an exception or returns `Y_OVERCURRENTLIMIT_INVALID`.

motor→get_starterTime() YMotor
motor→starterTime()YMotor get_starterTime

Returns the duration (in ms) during which the motor is driven at low frequency to help it start up.

YMotor **target** **get_starterTime**

Returns :

an integer corresponding to the duration (in ms) during which the motor is driven at low frequency to help it start up

On failure, throws an exception or returns `Y_STARTERTIME_INVALID`.

motor→keepALive()YMotor keepALive YMotor

Rearms the controller failsafe timer.

```
YMotor target keepALive
```

When the motor is running and the failsafe feature is active, this function should be called periodically to prove that the control process is running properly. Otherwise, the motor is automatically stopped after the specified timeout. Calling a motor *set* function implicitely rearms the failsafe timer.

motor→resetStatus()YMotor resetStatus YMotor

Reset the controller state to IDLE.

```
YMotor target resetStatus
```

This function must be invoked explicitely after any error condition is signaled.

motor→set_brakingForce() YMotor
motor→setBrakingForce()YMotor set_brakingForce

Changes immediately the braking force applied to the motor (in percents).

YMotor **target set_brakingForce newval**

The value 0 corresponds to no braking (free wheel). When the braking force is changed, the driving power is set to zero. The value is a percentage.

Parameters :

newval a floating point number corresponding to immediately the braking force applied to the motor (in percents)

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→set_cutOffVoltage() YMotor motor→setCutOffVoltage()YMotor set_cutOffVoltage

Changes the threshold voltage under which the controller automatically switches to error state and prevents further current draw.

```
YMotor target set_cutOffVoltage newval
```

This setting prevent damage to a battery that can occur when drawing current from an "empty" battery. Note that whatever the cutoff threshold, the controller switches to undervoltage error state if the power supply goes under 3V, even for a very brief time.

Parameters :

newval a floating point number corresponding to the threshold voltage under which the controller automatically switches to error state and prevents further current draw

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→set_drivingForce() YMotor
motor→setDrivingForce()YMotor set_drivingForce

Changes immediately the power sent to the motor.

```
YMotor target set_drivingForce newval
```

The value is a percentage between -100% to 100%. If you want go easy on your mechanics and avoid excessive current consumption, try to avoid brutal power changes. For example, immediate transition from forward full power to reverse full power is a very bad idea. Each time the driving power is modified, the braking power is set to zero.

Parameters :

newval a floating point number corresponding to immediately the power sent to the motor

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→set_failSafeTimeout() YMotor
motor→setFailSafeTimeout()YMotor
set_failSafeTimeout

Changes the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process.

```
YMotor target set_failSafeTimeout newval
```

When this delay has elapsed, the controller automatically stops the motor and switches to FAILSAFE error. Failsafe security is disabled when the value is zero.

Parameters :

newval an integer corresponding to the delay in milliseconds allowed for the controller to run autonomously without receiving any instruction from the control process

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→set_frequency() YMotor
motor→setFrequency()YMotor set_frequency

Changes the PWM frequency used to control the motor.

```
YMotor target set_frequency newval
```

Low frequency is usually more efficient and may help the motor to start, but an audible noise might be generated. A higher frequency reduces the noise, but more energy is converted into heat.

Parameters :

newval a floating point number corresponding to the PWM frequency used to control the motor

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→set_logicalName() YMotor motor→setLogicalName()YMotor set_logicalName

Changes the logical name of the motor.

```
YMotor 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 motor.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→set_overCurrentLimit() YMotor
motor→setOverCurrentLimit()YMotor
set_overCurrentLimit

Changes the current threshold (in mA) above which the controller automatically switches to error state.

```
YMotor target set_overCurrentLimit newval
```

A zero value means that there is no limit. Note that whatever the current limit is, the controller switches to OVERCURRENT status if the current goes above 32A, even for a very brief time.

Parameters :

newval an integer corresponding to the current threshold (in mA) above which the controller automatically switches to error state

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

motor→set_starterTime() YMotor
motor→setStarterTime()YMotor set_starterTime

Changes the duration (in ms) during which the motor is driven at low frequency to help it start up.

YMotor **target set_starterTime newval**

Parameters :

newval an integer corresponding to the duration (in ms) during which the motor is driven at low frequency to help it start up

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.28. Network function interface

YNetwork objects provide access to TCP/IP parameters of Yoctopuce modules that include a built-in network interface.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_network.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YNetwork = yoctolib.YNetwork;
php     require_once('yocto_network.php');
cpp     #include "yocto_network.h"
m       #import "yocto_network.h"
pas     uses yocto_network;
vb      yocto_network.vb
cs      yocto_network.cs
java    import com.yoctopuce.YoctoAPI.YNetwork;
py      from yocto_network import *
```

Global functions
yFindNetwork(**func**) Retrieves a network interface for a given identifier.
yFirstNetwork() Starts the enumeration of network interfaces currently accessible.
`YNetwork` methods
network→callbackLogin(**username**, **password**) Connects to the notification callback and saves the credentials required to log into it.
network→describe() Returns a short text that describes unambiguously the instance of the network interface in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
network→get_adminPassword() Returns a hash string if a password has been set for user "admin", or an empty string otherwise.
network→get_advertisedValue() Returns the current value of the network interface (no more than 6 characters).
network→get_callbackCredentials() Returns a hashed version of the notification callback credentials if set, or an empty string otherwise.
network→get_callbackEncoding() Returns the encoding standard to use for representing notification values.
network→get_callbackMaxDelay() Returns the maximum waiting time between two callback notifications, in seconds.
network→get_callbackMethod() Returns the HTTP method used to notify callbacks for significant state changes.
network→get_callbackMinDelay() Returns the minimum waiting time between two callback notifications, in seconds.
network→get_callbackUrl() Returns the callback URL to notify of significant state changes.
network→get_discoverable() Returns the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

network→get_errorMessage()

Returns the error message of the latest error with the network interface.

network→get_errorType()

Returns the numerical error code of the latest error with the network interface.

network→get_friendlyName()

Returns a global identifier of the network interface in the format `MODULE_NAME.FUNCTION_NAME`.

network→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

network→get_functionId()

Returns the hardware identifier of the network interface, without reference to the module.

network→get_hardwareId()

Returns the unique hardware identifier of the network interface in the form `SERIAL.FUNCTIONID`.

network→get_ipAddress()

Returns the IP address currently in use by the device.

network→get_logicalName()

Returns the logical name of the network interface.

network→get_macAddress()

Returns the MAC address of the network interface.

network→get_module()

Gets the `YModule` object for the device on which the function is located.

network→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

network→get_poeCurrent()

Returns the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps.

network→get_primaryDNS()

Returns the IP address of the primary name server to be used by the module.

network→get_readiness()

Returns the current established working mode of the network interface.

network→get_router()

Returns the IP address of the router on the device subnet (default gateway).

network→get_secondaryDNS()

Returns the IP address of the secondary name server to be used by the module.

network→get_subnetMask()

Returns the subnet mask currently used by the device.

network→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

network→get_userPassword()

Returns a hash string if a password has been set for "user" user, or an empty string otherwise.

network→get_wwwWatchdogDelay()

Returns the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

network→isOnline()

Checks if the network interface is currently reachable, without raising any error.

network→isOnline_async(**callback**, **context**)

Checks if the network interface is currently reachable, without raising any error (asynchronous version).

network→load(msValidity)

Preloads the network interface cache with a specified validity duration.

network→load_async(msValidity, callback, context)

Preloads the network interface cache with a specified validity duration (asynchronous version).

network→nextNetwork()

Continues the enumeration of network interfaces started using `yFirstNetwork()`.

network→ping(host)

Pings str_host to test the network connectivity.

network→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

network→set_adminPassword(newval)

Changes the password for the "admin" user.

network→set_callbackCredentials(newval)

Changes the credentials required to connect to the callback address.

network→set_callbackEncoding(newval)

Changes the encoding standard to use for representing notification values.

network→set_callbackMaxDelay(newval)

Changes the maximum waiting time between two callback notifications, in seconds.

network→set_callbackMethod(newval)

Changes the HTTP method used to notify callbacks for significant state changes.

network→set_callbackMinDelay(newval)

Changes the minimum waiting time between two callback notifications, in seconds.

network→set_callbackUrl(newval)

Changes the callback URL to notify significant state changes.

network→set_discoverable(newval)

Changes the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol).

network→set_logicalName(newval)

Changes the logical name of the network interface.

network→set_primaryDNS(newval)

Changes the IP address of the primary name server to be used by the module.

network→set_secondaryDNS(newval)

Changes the IP address of the secondary name server to be used by the module.

network→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

network→set_userPassword(newval)

Changes the password for the "user" user.

network→set_wwwWatchdogDelay(newval)

Changes the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

network→useDHCP(fallbackIpAddr, fallbackSubnetMaskLen, fallbackRouter)

Changes the configuration of the network interface to enable the use of an IP address received from a DHCP server.

network→useStaticIP(ipAddress, subnetMaskLen, router)

Changes the configuration of the network interface to use a static IP address.

network→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

network→callbackLogin()YNetwork callbackLogin YNetwork

Connects to the notification callback and saves the credentials required to log into it.

```
YNetwork target callbackLogin username password
```

The password is not stored into the module, only a hashed copy of the credentials are saved. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

username username required to log to the callback

password password required to log to the callback

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→get_adminPassword() YNetwork
network→adminPassword()YNetwork
get_adminPassword

Returns a hash string if a password has been set for user "admin", or an empty string otherwise.

YNetwork **target** **get_adminPassword**

Returns :

a string corresponding to a hash string if a password has been set for user "admin", or an empty string otherwise

On failure, throws an exception or returns `Y_ADMINPASSWORD_INVALID`.

network→get_advertisedValue() YNetwork
network→advertisedValue()YNetwork
get_advertisedValue

Returns the current value of the network interface (no more than 6 characters).

YNetwork **target** **get_advertisedValue**

Returns :
a string corresponding to the current value of the network interface (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

network→get_callbackCredentials() YNetwork
network→callbackCredentials()YNetwork
get_callbackCredentials

Returns a hashed version of the notification callback credentials if set, or an empty string otherwise.

YNetwork **target** **get_callbackCredentials**

Returns :
a string corresponding to a hashed version of the notification callback credentials if set, or an empty string otherwise

On failure, throws an exception or returns `Y_CALLBACKCREDENTIALS_INVALID`.

network→get_callbackEncoding() YNetwork
network→callbackEncoding()YNetwork
get_callbackEncoding

Returns the encoding standard to use for representing notification values.

YNetwork **target** **get_callbackEncoding**

Returns :

a value among `Y_CALLBACKENCODING_FORM`, `Y_CALLBACKENCODING_JSON`, `Y_CALLBACKENCODING_JSON_ARRAY`, `Y_CALLBACKENCODING_CSV` and `Y_CALLBACKENCODING_YOCTO_API` corresponding to the encoding standard to use for representing notification values

On failure, throws an exception or returns `Y_CALLBACKENCODING_INVALID`.

network→get_callbackMaxDelay() YNetwork
network→callbackMaxDelay()YNetwork
get_callbackMaxDelay

Returns the maximum waiting time between two callback notifications, in seconds.

YNetwork **target** **get_callbackMaxDelay**

Returns :
an integer corresponding to the maximum waiting time between two callback notifications, in seconds

On failure, throws an exception or returns `Y_CALLBACKMAXDELAY_INVALID`.

network→get_callbackMethod() YNetwork
network→callbackMethod()YNetwork
get_callbackMethod

Returns the HTTP method used to notify callbacks for significant state changes.

YNetwork **target** **get_callbackMethod**

Returns :

a value among `Y_CALLBACKMETHOD_POST`, `Y_CALLBACKMETHOD_GET` and `Y_CALLBACKMETHOD_PUT` corresponding to the HTTP method used to notify callbacks for significant state changes

On failure, throws an exception or returns `Y_CALLBACKMETHOD_INVALID`.

network→get_callbackMinDelay() YNetwork
network→callbackMinDelay()YNetwork
get_callbackMinDelay

Returns the minimum waiting time between two callback notifications, in seconds.

YNetwork **target** **get_callbackMinDelay**

Returns :
an integer corresponding to the minimum waiting time between two callback notifications, in seconds

On failure, throws an exception or returns `Y_CALLBACKMINDELAY_INVALID`.

network→get_callbackUrl() YNetwork
network→callbackUrl()YNetwork get_callbackUrl

Returns the callback URL to notify of significant state changes.

YNetwork **target** **get_callbackUrl**

Returns :
a string corresponding to the callback URL to notify of significant state changes

On failure, throws an exception or returns `Y_CALLBACKURL_INVALID`.

network→get_discoverable() YNetwork network→discoverable()YNetwork 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).

YNetwork **target** **get_discoverable**

Returns :

either Y_DISCOVERABLE_FALSE or Y_DISCOVERABLE_TRUE, according to the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol)

On failure, throws an exception or returns Y_DISCOVERABLE_INVALID.

network→get_ipAddress() YNetwork network→ipAddress()YNetwork get_ipAddress

Returns the IP address currently in use by the device.

```
YNetwork target get_ipAddress
```

The address may have been configured statically, or provided by a DHCP server.

Returns :
a string corresponding to the IP address currently in use by the device

On failure, throws an exception or returns `Y_IPADDRESS_INVALID`.

network→get_logicalName() YNetwork network→logicalName()YNetwork get_logicalName

Returns the logical name of the network interface.

YNetwork **target** **get_logicalName**

Returns :
a string corresponding to the logical name of the network interface.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

network→get_macAddress() YNetwork network→macAddress()YNetwork get_macAddress

Returns the MAC address of the network interface.

```
YNetwork target get_macAddress
```

The MAC address is also available on a sticker on the module, in both numeric and barcode forms.

Returns :
a string corresponding to the MAC address of the network interface

On failure, throws an exception or returns `Y_MACADDRESS_INVALID`.

network→get_poeCurrent() YNetwork network→poeCurrent()YNetwork get_poeCurrent

Returns the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps.

YNetwork **target** **get_poeCurrent**

The current consumption is measured after converting PoE source to 5 Volt, and should never exceed 1800 mA.

Returns :

an integer corresponding to the current consumed by the module from Power-over-Ethernet (PoE), in milli-amps

On failure, throws an exception or returns `Y_POECURRENT_INVALID`.

network→get_primaryDNS() YNetwork network→primaryDNS()YNetwork get_primaryDNS

Returns the IP address of the primary name server to be used by the module.

YNetwork **target get_primaryDNS**

Returns :

a string corresponding to the IP address of the primary name server to be used by the module

On failure, throws an exception or returns `Y_PRIMARYDNS_INVALID`.

network→get_readiness() YNetwork
network→readiness()YNetwork get_readiness

Returns the current established working mode of the network interface.

YNetwork **target** **get_readiness**

Level zero (DOWN_0) means that no hardware link has been detected. Either there is no signal on the network cable, or the selected wireless access point cannot be detected. Level 1 (LIVE_1) is reached when the network is detected, but is not yet connected. For a wireless network, this shows that the requested SSID is present. Level 2 (LINK_2) is reached when the hardware connection is established. For a wired network connection, level 2 means that the cable is attached at both ends. For a connection to a wireless access point, it shows that the security parameters are properly configured. For an ad-hoc wireless connection, it means that there is at least one other device connected on the ad-hoc network. Level 3 (DHCP_3) is reached when an IP address has been obtained using DHCP. Level 4 (DNS_4) is reached when the DNS server is reachable on the network. Level 5 (WWW_5) is reached when global connectivity is demonstrated by properly loading the current time from an NTP server.

Returns :

a value among `Y_READINESS_DOWN`, `Y_READINESS_EXISTS`, `Y_READINESS_LINKED`, `Y_READINESS_LAN_OK` and `Y_READINESS_WWW_OK` corresponding to the current established working mode of the network interface

On failure, throws an exception or returns `Y_READINESS_INVALID`.

network→get_router() YNetwork
network→router()YNetwork get_router

Returns the IP address of the router on the device subnet (default gateway).

YNetwork **target** **get_router**

Returns :

a string corresponding to the IP address of the router on the device subnet (default gateway)

On failure, throws an exception or returns `Y_ROUTER_INVALID`.

network→get_secondaryDNS() YNetwork
network→secondaryDNS()YNetwork
get_secondaryDNS

Returns the IP address of the secondary name server to be used by the module.

YNetwork **target get_secondaryDNS**

Returns :
a string corresponding to the IP address of the secondary name server to be used by the module

On failure, throws an exception or returns `Y_SECONDARYDNS_INVALID`.

network→get_subnetMask() YNetwork network→subnetMask()YNetwork get_subnetMask

Returns the subnet mask currently used by the device.

YNetwork **target** **get_subnetMask**

Returns :

a string corresponding to the subnet mask currently used by the device

On failure, throws an exception or returns `Y_SUBNETMASK_INVALID`.

network→get_userPassword() YNetwork
network→userPassword()YNetwork
get_userPassword

Returns a hash string if a password has been set for "user" user, or an empty string otherwise.

YNetwork **target** **get_userPassword**

Returns :
a string corresponding to a hash string if a password has been set for "user" user, or an empty string otherwise

On failure, throws an exception or returns `Y_USERPASSWORD_INVALID`.

network→get_wwwWatchdogDelay() YNetwork
network→wwwWatchdogDelay()YNetwork
get_wwwWatchdogDelay

Returns the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

YNetwork **target** **get_wwwWatchdogDelay**

A zero value disables automated reboot in case of Internet connectivity loss.

Returns :

an integer corresponding to the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity

On failure, throws an exception or returns `Y_WWWWATCHDOGDELAY_INVALID`.

network→ping()YNetwork ping YNetwork

Pings str_host to test the network connectivity.

```
YNetwork target ping host
```

Sends four ICMP ECHO_REQUEST requests from the module to the target str_host. This method returns a string with the result of the 4 ICMP ECHO_REQUEST requests.

Parameters :

host the hostname or the IP address of the target

Returns :

a string with the result of the ping.

network→set_adminPassword() YNetwork
network→setAdminPassword()YNetwork
set_adminPassword

Changes the password for the "admin" user.

YNetwork **target** **set_adminPassword** **newval**

This password becomes instantly required to perform any change of the module state. If the specified value is an empty string, a password is not required anymore. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the password for the "admin" user

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackCredentials() YNetwork
network→setCallbackCredentials()YNetwork
set_callbackCredentials

Changes the credentials required to connect to the callback address.

YNetwork **target set_callbackCredentials newval**

The credentials must be provided as returned by function `get_callbackCredentials`, in the form `username:hash`. The method used to compute the hash varies according to the 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()YNetwork
set_callbackEncoding

Changes the encoding standard to use for representing notification values.

YNetwork **target** **set_callbackEncoding** **newval**

Parameters :

newval a value among `Y_CALLBACKENCODING_FORM`, `Y_CALLBACKENCODING_JSON`, `Y_CALLBACKENCODING_JSON_ARRAY`, `Y_CALLBACKENCODING_CSV` and `Y_CALLBACKENCODING_YOCTO_API` corresponding to the encoding standard to use for representing notification values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackMaxDelay() YNetwork
network→setCallbackMaxDelay()YNetwork
set_callbackMaxDelay

Changes the maximum waiting time between two callback notifications, in seconds.

YNetwork **target set_callbackMaxDelay newval**

Parameters :

newval an integer corresponding to the maximum waiting time between two callback notifications, in seconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackMethod() YNetwork
network→setCallbackMethod()YNetwork
set_callbackMethod

Changes the HTTP method used to notify callbacks for significant state changes.

YNetwork **target** **set_callbackMethod** **newval**

Parameters :

newval a value among `Y_CALLBACKMETHOD_POST`, `Y_CALLBACKMETHOD_GET` and `Y_CALLBACKMETHOD_PUT` corresponding to the HTTP method used to notify callbacks for significant state changes

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackMinDelay() YNetwork
network→setCallbackMinDelay()YNetwork
set_callbackMinDelay

Changes the minimum waiting time between two callback notifications, in seconds.

YNetwork **target** **set_callbackMinDelay** **newval**

Parameters :

newval an integer corresponding to the minimum waiting time between two callback notifications, in seconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_callbackUrl() YNetwork
network→setCallbackUrl()YNetwork set_callbackUrl

Changes the callback URL to notify significant state changes.

```
YNetwork target set_callbackUrl newval
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the callback URL to notify significant state changes

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_discoverable() YNetwork
network→setDiscoverable()YNetwork
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).

YNetwork **target** **set_discoverable** **newval**

Parameters :

newval either `Y_DISCOVERABLE_FALSE` or `Y_DISCOVERABLE_TRUE`, according to the activation state of the multicast announce protocols to allow easy discovery of the module in the network neighborhood (uPnP/Bonjour protocol)

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_logicalName() YNetwork
network→setLogicalName()YNetwork
set_logicalName

Changes the logical name of the network interface.

YNetwork **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 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()YNetwork
set_primaryDNS

Changes the IP address of the primary name server to be used by the module.

```
YNetwork target set_primaryDNS newval
```

When using DHCP, if a value is specified, it overrides the value received from the DHCP server. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

newval a string corresponding to the IP address of the primary name server to be used by the module

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_secondaryDNS() YNetwork
network→setSecondaryDNS()YNetwork
set_secondaryDNS

Changes the IP address of the secondary name server to be used by the module.

YNetwork **target set_secondaryDNS newval**

When using DHCP, if a value is specified, it overrides the value received from the DHCP server. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

newval a string corresponding to the IP address of the secondary name server to be used by the module

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_userPassword() YNetwork
network→setUserPassword()YNetwork
set_userPassword

Changes the password for the "user" user.

```
YNetwork target set_userPassword newval
```

This password becomes instantly required to perform any use of the module. If the specified value is an empty string, a password is not required anymore. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the password for the "user" user

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→set_wwwWatchdogDelay() YNetwork
network→setWwwWatchdogDelay()YNetwork
set_wwwWatchdogDelay

Changes the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity.

YNetwork **target set_wwwWatchdogDelay newval**

A zero value disables automated reboot in case of Internet connectivity loss. The smallest valid non-zero timeout is 90 seconds.

Parameters :

newval an integer corresponding to the allowed downtime of the WWW link (in seconds) before triggering an automated reboot to try to recover Internet connectivity

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

network→useDHCP()YNetwork useDHCP YNetwork

Changes the configuration of the network interface to enable the use of an IP address received from a DHCP server.

YNetwork **target useDHCP fallbackIpAddr fallbackSubnetMaskLen fallbackRouter**

Until an address is received from a DHCP server, the module uses the IP parameters specified to this function. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

fallbackIpAddr	fallback IP address, to be used when no DHCP reply is received
fallbackSubnetMaskLen	fallback subnet mask length when no DHCP reply is received, as an integer (eg. 24 means 255.255.255.0)
fallbackRouter	fallback router IP address, to be used when no DHCP reply is received

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

network→useStaticIP()YNetwork useStaticIP YNetwork

Changes the configuration of the network interface to use a static IP address.

YNetwork **target useStaticIP ipAddress subnetMaskLen router**

Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

ipAddress	device IP address
subnetMaskLen	subnet mask length, as an integer (eg. 24 means 255.255.255.0)
router	router IP address (default gateway)

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

3.29. OS control

The OScontrol object allows some control over the operating system running a VirtualHub. OsControl is available on the VirtualHub software only. This feature must be activated at the VirtualHub start up with -o option.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_oscontrol.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YOsControl = yoctolib.YOsControl;
php     require_once('yocto_oscontrol.php');
cpp     #include "yocto_oscontrol.h"
m       #import "yocto_oscontrol.h"
pas     uses yocto_oscontrol;
vb      yocto_oscontrol.vb
cs      yocto_oscontrol.cs
java    import com.yoctopuce.YoctoAPI.YOsControl;
py      from yocto_oscontrol import *
```

Global functions

yFindOsControl(**func**)

Retrieves OS control for a given identifier.

yFirstOsControl()

Starts the enumeration of OS control currently accessible.

`YOsControl` methods

oscontrol→describe()

Returns a short text that describes unambiguously the instance of the OS control in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

oscontrol→get_advertisedValue()

Returns the current value of the OS control (no more than 6 characters).

oscontrol→get_errorMessage()

Returns the error message of the latest error with the OS control.

oscontrol→get_errorType()

Returns the numerical error code of the latest error with the OS control.

oscontrol→get_friendlyName()

Returns a global identifier of the OS control in the format `MODULE_NAME.FUNCTION_NAME`.

oscontrol→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

oscontrol→get_functionId()

Returns the hardware identifier of the OS control, without reference to the module.

oscontrol→get_hardwareId()

Returns the unique hardware identifier of the OS control in the form `SERIAL.FUNCTIONID`.

oscontrol→get_logicalName()

Returns the logical name of the OS control.

oscontrol→get_module()

Gets the `YModule` object for the device on which the function is located.

oscontrol→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

oscontrol→get_shutdownCountdown()

Returns the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled.

oscontrol→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

oscontrol→isOnline()

Checks if the OS control is currently reachable, without raising any error.

oscontrol→isOnline_async(**callback**, **context**)

Checks if the OS control is currently reachable, without raising any error (asynchronous version).

oscontrol→load(**msValidity**)

Preloads the OS control cache with a specified validity duration.

oscontrol→load_async(**msValidity**, **callback**, **context**)

Preloads the OS control cache with a specified validity duration (asynchronous version).

oscontrol→nextOsControl()

Continues the enumeration of OS control started using `yFirstOsControl()`.

oscontrol→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

oscontrol→set_logicalName(**newval**)

Changes the logical name of the OS control.

oscontrol→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

oscontrol→shutdown(**secBeforeShutDown**)

Schedules an OS shutdown after a given number of seconds.

oscontrol→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

oscontrol→get_advertisedValue() YOsControl
oscontrol→advertisedValue()YOsControl
get_advertisedValue

Returns the current value of the OS control (no more than 6 characters).

```
YOsControl target get_advertisedValue
```

Returns :
a string corresponding to the current value of the OS control (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

oscontrol→get_logicalName() YOsControl
oscontrol→logicalName()YOsControl
get_logicalName

Returns the logical name of the OS control.

YOsControl **target** **get_logicalName**

Returns :
a string corresponding to the logical name of the OS control.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

oscontrol→get_shutdownCountdown() YOsControl
oscontrol→shutdownCountdown()YOsControl
get_shutdownCountdown

Returns the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled.

YOsControl **target get_shutdownCountdown**

Returns :

an integer corresponding to the remaining number of seconds before the OS shutdown, or zero when no shutdown has been scheduled

On failure, throws an exception or returns `Y_SHUTDOWNCOUNTDOWN_INVALID`.

oscontrol→set_logicalName() YOsControl
oscontrol→setLogicalName()YOsControl
set_logicalName

Changes the logical name of the OS control.

```
YOsControl 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 OS control.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

oscontrol→shutdown()YOsControl shutdown YOsControl

Schedules an OS shutdown after a given number of seconds.

YOsControl **target shutdown secBeforeShutDown**

Parameters :

secBeforeShutDown number of seconds before shutdown

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

3.30. Power function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
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');
cpp     #include "yocto_power.h"
m       #import "yocto_power.h"
pas     uses yocto_power;
vb      yocto_power.vb
cs      yocto_power.cs
java    import com.yoctopuce.YoctoAPI.YPower;
py      from yocto_power import *
```

Global functions
yFindPower(**func**) Retrieves a electrical power sensor for a given identifier.
yFirstPower() Starts the enumeration of electrical power sensors currently accessible.
YPower methods
power→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
power→describe() Returns a short text that describes unambiguously the instance of the electrical power sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
power→get_advertisedValue() Returns the current value of the electrical power sensor (no more than 6 characters).
power→get_cosPhi() Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).
power→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in Watt, as a floating point number.
power→get_currentValue() Returns the current value of the electrical power, in Watt, as a floating point number.
power→get_errorMessage() Returns the error message of the latest error with the electrical power sensor.
power→get_errorType() Returns the numerical error code of the latest error with the electrical power sensor.
power→get_friendlyName() Returns a global identifier of the electrical power sensor in the format `MODULE_NAME.FUNCTION_NAME`.
power→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
power→get_functionId()

Returns the hardware identifier of the electrical power sensor, without reference to the module.

power→get_hardwareId()

Returns the unique hardware identifier of the electrical power sensor in the form `SERIAL.FUNCTIONID`.

power→get_highestValue()

Returns the maximal value observed for the electrical power since the device was started.

power→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

power→get_logicalName()

Returns the logical name of the electrical power sensor.

power→get_lowestValue()

Returns the minimal value observed for the electrical power since the device was started.

power→get_meter()

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.

power→get_meterTimer()

Returns the elapsed time since last energy counter reset, in seconds.

power→get_module()

Gets the `YModule` object for the device on which the function is located.

power→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

power→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

power→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

power→get_resolution()

Returns the resolution of the measured values.

power→get_unit()

Returns the measuring unit for the electrical power.

power→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

power→isOnline()

Checks if the electrical power sensor is currently reachable, without raising any error.

power→isOnline_async(**callback**, **context**)

Checks if the electrical power sensor is currently reachable, without raising any error (asynchronous version).

power→load(**msValidity**)

Preloads the electrical power sensor cache with a specified validity duration.

power→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

power→load_async(**msValidity**, **callback**, **context**)

Preloads the electrical power sensor cache with a specified validity duration (asynchronous version).

power→nextPower()

Continues the enumeration of electrical power sensors started using `yFirstPower()`.

power→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

power→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

power→reset()

Resets the energy counter.

power→set_highestValue(**newval**)

Changes the recorded maximal value observed.

power→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

power→set_logicalName(**newval**)

Changes the logical name of the electrical power sensor.

power→set_lowestValue(**newval**)

Changes the recorded minimal value observed.

power→set_reportFrequency(**newval**)

Changes the timed value notification frequency for this function.

power→set_resolution(**newval**)

Changes the resolution of the measured physical values.

power→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

power→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

power→calibrateFromPoints()YPower YPower calibrateFromPoints

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

YPower **target calibrateFromPoints rawValues refValues**

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→get_advertisedValue() YPower
power→advertisedValue()YPower
get_advertisedValue

Returns the current value of the electrical power sensor (no more than 6 characters).

YPower **target** **get_advertisedValue**

Returns :
a string corresponding to the current value of the electrical power sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

power→get_cosPhi() YPower power→cosPhi()YPower get_cosPhi

Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).

```
YPower target get_cosPhi
```

Returns :

a floating point number corresponding to the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA)

On failure, throws an exception or returns `Y_COSPHI_INVALID`.

power→get_currentRawValue() YPower
power→currentRawValue()YPower
get_currentRawValue

Returns the uncalibrated, unrounded raw value returned by the sensor, in Watt, as a floating point number.

YPower **target** **get_currentRawValue**

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in Watt, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

power→get_currentValue() YPower
power→currentValue()YPower get_currentValue

Returns the current value of the electrical power, in Watt, as a floating point number.

```
YPower target get_currentValue
```

Returns :

a floating point number corresponding to the current value of the electrical power, in Watt, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

power→get_highestValue() YPower power→highestValue()YPower get_highestValue

Returns the maximal value observed for the electrical power since the device was started.

YPower **target get_highestValue**

Returns :

a floating point number corresponding to the maximal value observed for the electrical power since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

power→get_logFrequency() YPower
power→logFrequency()YPower get_logFrequency

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

YPower **target get_logFrequency**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

power→get_logicalName() YPower power→logicalName()YPower get_logicalName

Returns the logical name of the electrical power sensor.

YPower **target get_logicalName**

Returns :
a string corresponding to the logical name of the electrical power sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

power→get_lowestValue() YPower power→lowestValue()YPower get_lowestValue

Returns the minimal value observed for the electrical power since the device was started.

YPower **target** **get_lowestValue**

Returns :

a floating point number corresponding to the minimal value observed for the electrical power since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

power→get_meter() YPower
power→meter()YPower get_meter

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time.

```
YPower target get_meter
```

Note that this counter is reset at each start of the device.

Returns :

a floating point number corresponding to the energy counter, maintained by the wattmeter by integrating the power consumption over time

On failure, throws an exception or returns `Y_METER_INVALID`.

power→get_meterTimer() YPower
power→meterTimer()YPower get_meterTimer

Returns the elapsed time since last energy counter reset, in seconds.

YPower **target get_meterTimer**

Returns :
an integer corresponding to the elapsed time since last energy counter reset, in seconds

On failure, throws an exception or returns `Y_METERTIMER_INVALID`.

power→get_recordedData() YPower power→recordedData()YPower get_recordedData

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
YPower target get_recordedData startTime endTime
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

power→get_reportFrequency() YPower
power→reportFrequency()YPower
get_reportFrequency

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

YPower **target get_reportFrequency**

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

power→get_resolution() YPower power→resolution()YPower get_resolution

Returns the resolution of the measured values.

```
YPower target get_resolution
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

power→get_unit() YPower power→unit()YPower get_unit

Returns the measuring unit for the electrical power.

```
YPower target get_unit
```

Returns :

a string corresponding to the measuring unit for the electrical power

On failure, throws an exception or returns `Y_UNIT_INVALID`.

power→loadCalibrationPoints()YPower loadCalibrationPoints

YPower

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

YPower **target loadCalibrationPoints rawValues refValues**

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→reset()YPower reset YPower

Resets the energy counter.

```
YPower target reset
```

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_highestValue() YPower
power→setHighestValue()YPower set_highestValue

Changes the recorded maximal value observed.

```
YPower target set_highestValue newval
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_logFrequency() YPower
power→setLogFrequency()YPower
set_logFrequency

Changes the datalogger recording frequency for this function.

```
YPower target set_logFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_logicalName() YPower power→setLogicalName()YPower set_logicalName

Changes the logical name of the electrical power sensor.

```
YPower 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 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()YPower set_lowestValue

Changes the recorded minimal value observed.

```
YPower target set_lowestValue newval
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_reportFrequency() YPower
power→setReportFrequency()YPower
set_reportFrequency

Changes the timed value notification frequency for this function.

```
YPower target set_reportFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_resolution() YPower power→setResolution()YPower set_resolution

Changes the resolution of the measured physical values.

```
YPower target set_resolution newval
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.31. Pressure function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_pressure.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YPressure = yoctolib.YPressure;
php     require_once('yocto_pressure.php');
cpp     #include "yocto_pressure.h"
m       #import "yocto_pressure.h"
pas     uses yocto_pressure;
vb      yocto_pressure.vb
cs      yocto_pressure.cs
java    import com.yoctopuce.YoctoAPI.YPressure;
py      from yocto_pressure import *
```

Global functions
yFindPressure(**func**) Retrieves a pressure sensor for a given identifier.
yFirstPressure() Starts the enumeration of pressure sensors currently accessible.
YPressure methods
pressure→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
pressure→describe() Returns a short text that describes unambiguously the instance of the pressure sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
pressure→get_advertisedValue() Returns the current value of the pressure sensor (no more than 6 characters).
pressure→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in millibar (hPa), as a floating point number.
pressure→get_currentValue() Returns the current value of the pressure, in millibar (hPa), as a floating point number.
pressure→get_errorMessage() Returns the error message of the latest error with the pressure sensor.
pressure→get_errorType() Returns the numerical error code of the latest error with the pressure sensor.
pressure→get_friendlyName() Returns a global identifier of the pressure sensor in the format `MODULE_NAME.FUNCTION_NAME`.
pressure→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
pressure→get_functionId() Returns the hardware identifier of the pressure sensor, without reference to the module.
pressure→get_hardwareId()

Returns the unique hardware identifier of the pressure sensor in the form `SERIAL.FUNCTIONID`.

pressure→get_highestValue()

Returns the maximal value observed for the pressure since the device was started.

pressure→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

pressure→get_logicalName()

Returns the logical name of the pressure sensor.

pressure→get_lowestValue()

Returns the minimal value observed for the pressure since the device was started.

pressure→get_module()

Gets the `YModule` object for the device on which the function is located.

pressure→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

pressure→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

pressure→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

pressure→get_resolution()

Returns the resolution of the measured values.

pressure→get_unit()

Returns the measuring unit for the pressure.

pressure→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

pressure→isOnline()

Checks if the pressure sensor is currently reachable, without raising any error.

pressure→isOnline_async(**callback**, **context**)

Checks if the pressure sensor is currently reachable, without raising any error (asynchronous version).

pressure→load(**msValidity**)

Preloads the pressure sensor cache with a specified validity duration.

pressure→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

pressure→load_async(**msValidity**, **callback**, **context**)

Preloads the pressure sensor cache with a specified validity duration (asynchronous version).

pressure→nextPressure()

Continues the enumeration of pressure sensors started using `yFirstPressure()`.

pressure→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

pressure→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

pressure→set_highestValue(**newval**)

Changes the recorded maximal value observed.

pressure→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

pressure→set_logicalName(**newval**)

Changes the logical name of the pressure sensor.

pressure→set_lowestValue(newval)

Changes the recorded minimal value observed.

pressure→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

pressure→set_resolution(newval)

Changes the resolution of the measured physical values.

pressure→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

pressure→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

pressure→calibrateFromPoints()YPressure calibrateFromPoints YPressure

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

YPressure **target calibrateFromPoints rawValues refValues**

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→get_advertisedValue() YPressure
pressure→advertisedValue()YPressure
get_advertisedValue

Returns the current value of the pressure sensor (no more than 6 characters).

```
YPressure target get_advertisedValue
```

Returns :
a string corresponding to the current value of the pressure sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

pressure→get_currentRawValue() YPressure
pressure→currentRawValue()YPressure
get_currentRawValue

Returns the uncalibrated, unrounded raw value returned by the sensor, in millibar (hPa), as a floating point number.

YPressure **target get_currentRawValue**

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in millibar (hPa), as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

pressure→get_currentValue() YPressure
pressure→currentValue()YPressure
get_currentValue

Returns the current value of the pressure, in millibar (hPa), as a floating point number.

YPressure **target get_currentValue**

Returns :
a floating point number corresponding to the current value of the pressure, in millibar (hPa), as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

pressure→get_highestValue() YPressure
pressure→highestValue()YPressure
get_highestValue

Returns the maximal value observed for the pressure since the device was started.

YPressure **target** **get_highestValue**

Returns :
a floating point number corresponding to the maximal value observed for the pressure since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

pressure→get_logFrequency() YPressure
pressure→logFrequency()YPressure
get_logFrequency

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

YPressure **target get_logFrequency**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

pressure→get_logicalName() YPressure
pressure→logicalName()YPressure get_logicalName

Returns the logical name of the pressure sensor.

```
YPressure target get_logicalName
```

Returns :
a string corresponding to the logical name of the pressure sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

pressure→get_lowestValue() YPressure
pressure→lowestValue()YPressure get_lowestValue

Returns the minimal value observed for the pressure since the device was started.

```
YPressure target get_lowestValue
```

Returns :

a floating point number corresponding to the minimal value observed for the pressure since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

pressure→get_recordedData() YPressure
pressure→recordedData()YPressure
get_recordedData

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YPressure **target get_recordedData startTime endTime**

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

pressure→get_reportFrequency() YPressure
pressure→reportFrequency()YPressure
get_reportFrequency

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

YPressure **target get_reportFrequency**

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

pressure→get_resolution() YPressure
pressure→resolution()YPressure get_resolution

Returns the resolution of the measured values.

```
YPressure target get_resolution
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

pressure→get_unit() YPressure
pressure→unit()YPressure get_unit

Returns the measuring unit for the pressure.

```
YPressure target get_unit
```

Returns :
a string corresponding to the measuring unit for the pressure

On failure, throws an exception or returns `Y_UNIT_INVALID`.

pressure→loadCalibrationPoints()YPressure YPressure loadCalibrationPoints

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

YPressure **target loadCalibrationPoints rawValues refValues**

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→set_highestValue() YPressure
pressure→setHighestValue()YPressure
set_highestValue

Changes the recorded maximal value observed.

```
YPressure target set_highestValue newval
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→set_logFrequency() YPressure
pressure→setLogFrequency()YPressure
set_logFrequency

Changes the datalogger recording frequency for this function.

```
YPressure target set_logFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→set_logicalName() YPressure
pressure→setLogicalName()YPressure
set_logicalName

Changes the logical name of the pressure sensor.

```
YPressure 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 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()YPressure
set_lowestValue

Changes the recorded minimal value observed.

YPressure **target set_lowestValue newval**

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→set_reportFrequency() YPressure
pressure→setReportFrequency()YPressure
set_reportFrequency

Changes the timed value notification frequency for this function.

```
YPressure target set_reportFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pressure→set_resolution() YPressure pressure→setResolution()YPressure set_resolution

Changes the resolution of the measured physical values.

```
YPressure target set_resolution newval
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.32. PwmInput function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_pwminput.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YPwmInput = yoctolib.YPwmInput;
php     require_once('yocto_pwminput.php');
cpp     #include "yocto_pwminput.h"
m       #import "yocto_pwminput.h"
pas     uses yocto_pwminput;
vb      yocto_pwminput.vb
cs      yocto_pwminput.cs
java    import com.yoctopuce.YoctoAPI.YPwmInput;
py      from yocto_pwminput import *
```

Global functions
yFindPwmInput(**func**) Retrieves a voltage sensor for a given identifier.
yFirstPwmInput() Starts the enumeration of voltage sensors currently accessible.
`YPwmInput` methods
pwminput→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
pwminput→describe() Returns a short text that describes unambiguously the instance of the voltage sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
pwminput→get_advertisedValue() Returns the current value of the voltage sensor (no more than 6 characters).
pwminput→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number.
pwminput→get_currentValue() Returns the current value of PwmInput feature as a floating point number.
pwminput→get_dutyCycle() Returns the PWM duty cycle, in per cents.
pwminput→get_errorMessage() Returns the error message of the latest error with the voltage sensor.
pwminput→get_errorType() Returns the numerical error code of the latest error with the voltage sensor.
pwminput→get_frequency() Returns the PWM frequency in Hz.
pwminput→get_friendlyName() Returns a global identifier of the voltage sensor in the format `MODULE_NAME.FUNCTION_NAME`.
pwminput→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

pwminput→get_functionId()

Returns the hardware identifier of the voltage sensor, without reference to the module.

pwminput→get_hardwareId()

Returns the unique hardware identifier of the voltage sensor in the form `SERIAL.FUNCTIONID`.

pwminput→get_highestValue()

Returns the maximal value observed for the voltage since the device was started.

pwminput→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

pwminput→get_logicalName()

Returns the logical name of the voltage sensor.

pwminput→get_lowestValue()

Returns the minimal value observed for the voltage since the device was started.

pwminput→get_module()

Gets the `YModule` object for the device on which the function is located.

pwminput→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

pwminput→get_period()

Returns the PWM period in milliseconds.

pwminput→get_pulseCounter()

Returns the pulse counter value.

pwminput→get_pulseDuration()

Returns the PWM pulse length in milliseconds, as a floating point number.

pwminput→get_pulseTimer()

Returns the timer of the pulses counter (ms)

pwminput→get_pwmReportMode()

Returns the parameter (frequency/duty cycle, pulse width, edges count) returned by the get_currentValue function and callbacks.

pwminput→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

pwminput→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

pwminput→get_resolution()

Returns the resolution of the measured values.

pwminput→get_unit()

Returns the measuring unit for the values returned by get_currentValue and callbacks.

pwminput→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

pwminput→isOnline()

Checks if the voltage sensor is currently reachable, without raising any error.

pwminput→isOnline_async(**callback**, **context**)

Checks if the voltage sensor is currently reachable, without raising any error (asynchronous version).

pwminput→load(**msValidity**)

Preloads the voltage sensor cache with a specified validity duration.

pwminput→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

pwminput→load_async(msValidity, callback, context)

Preloads the voltage sensor cache with a specified validity duration (asynchronous version).

pwminput→nextPwmInput()

Continues the enumeration of voltage sensors started using `yFirstPwmInput()`.

pwminput→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

pwminput→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

pwminput→resetCounter()

Returns the pulse counter value as well as his timer

pwminput→set_highestValue(newval)

Changes the recorded maximal value observed.

pwminput→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

pwminput→set_logicalName(newval)

Changes the logical name of the voltage sensor.

pwminput→set_lowestValue(newval)

Changes the recorded minimal value observed.

pwminput→set_pwmReportMode(newval)

Modify the parameter type(frequency/duty cycle, pulse width ou edge count) returned by the get_currentValue function and callbacks.

pwminput→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

pwminput→set_resolution(newval)

Changes the resolution of the measured physical values.

pwminput→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

pwminput→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

pwminput→calibrateFromPoints()YPwmInput YPwmInput calibrateFromPoints

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

YPwmInput **target calibrateFromPoints rawValues refValues**

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwminput→get_advertisedValue() YPwmInput
pwminput→advertisedValue()YPwmInput
get_advertisedValue

Returns the current value of the voltage sensor (no more than 6 characters).

YPwmInput **target** **get_advertisedValue**

Returns :
a string corresponding to the current value of the voltage sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

pwminput→get_currentRawValue() YPwmInput
pwminput→currentRawValue()YPwmInput
get_currentRawValue

Returns the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number.

YPwmInput **target get_currentRawValue**

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

pwminput→get_currentValue() YPwmInput
pwminput→currentValue()YPwmInput
get_currentValue

Returns the current value of PwmInput feature as a floating point number.

```
YPwmInput target get_currentValue
```

Depending on the pwmReportMode setting, this can be the frequency, in Hz, the duty cycle in % or the pulse length.

Returns :
a floating point number corresponding to the current value of PwmInput feature as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

pwminput→get_dutyCycle() YPwmInput
pwminput→dutyCycle()YPwmInput get_dutyCycle

Returns the PWM duty cycle, in per cents.

YPwmInput **target get_dutyCycle**

Returns :

a floating point number corresponding to the PWM duty cycle, in per cents

On failure, throws an exception or returns `Y_DUTYCYCLE_INVALID`.

pwminput→get_frequency() YPwmInput
pwminput→frequency()YPwmInput get_frequency

Returns the PWM frequency in Hz.

YPwmInput **target** **get_frequency**

Returns :
a floating point number corresponding to the PWM frequency in Hz

On failure, throws an exception or returns `Y_FREQUENCY_INVALID`.

pwminput→get_highestValue() YPwmInput
pwminput→highestValue()YPwmInput
get_highestValue

Returns the maximal value observed for the voltage since the device was started.

YPwmInput **target get_highestValue**

Returns :
a floating point number corresponding to the maximal value observed for the voltage since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

pwminput→get_logFrequency() YPwmInput
pwminput→logFrequency()YPwmInput
get_logFrequency

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

YPwmInput **target** **get_logFrequency**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

pwminput→get_logicalName() YPwmInput
pwminput→logicalName()YPwmInput
get_logicalName

Returns the logical name of the voltage sensor.

YPwmInput **target get_logicalName**

Returns :
a string corresponding to the logical name of the voltage sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

pwminput→get_lowestValue() YPwmInput
pwminput→lowestValue()YPwmInput
get_lowestValue

Returns the minimal value observed for the voltage since the device was started.

YPwmInput **target** **get_lowestValue**

Returns :
a floating point number corresponding to the minimal value observed for the voltage since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

pwminput→get_period() YPwmInput
pwminput→period()YPwmInput get_period

Returns the PWM period in milliseconds.

YPwmInput **target get_period**

Returns :
a floating point number corresponding to the PWM period in milliseconds

On failure, throws an exception or returns `Y_PERIOD_INVALID`.

pwminput→get_pulseCounter() YPwmInput
pwminput→pulseCounter()YPwmInput
get_pulseCounter

Returns the pulse counter value.

```
YPwmInput target get_pulseCounter
```

Actually that counter is incremented twice per period. That counter is limited to 1 billions

Returns :
an integer corresponding to the pulse counter value

On failure, throws an exception or returns `Y_PULSECOUNTER_INVALID`.

pwminput→get_pulseDuration() YPwmInput
pwminput→pulseDuration()YPwmInput
get_pulseDuration

Returns the PWM pulse length in milliseconds, as a floating point number.

YPwmInput **target get_pulseDuration**

Returns :
a floating point number corresponding to the PWM pulse length in milliseconds, as a floating point number

On failure, throws an exception or returns `Y_PULSEDURATION_INVALID`.

pwminput→get_pwmReportMode() YPwmInput
pwminput→pwmReportMode()YPwmInput
get_pwmReportMode

Returns the parameter (frequency/duty cycle, pulse width, edges count) returned by the get_currentValue function and callbacks.

```
YPwmInput target get_pwmReportMode
```

Attention

Returns :

a value among `Y_PWMREPORTMODE_PWM_DUTYCYCLE`, `Y_PWMREPORTMODE_PWM_FREQUENCY`, `Y_PWMREPORTMODE_PWM_PULSEDURATION` and `Y_PWMREPORTMODE_PWM_EDGECOUNT` corresponding to the parameter (frequency/duty cycle, pulse width, edges count) returned by the get_currentValue function and callbacks

On failure, throws an exception or returns `Y_PWMREPORTMODE_INVALID`.

pwminput→get_recordedData() YPwmInput
pwminput→recordedData()YPwmInput
get_recordedData

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YPwmInput **target get_recordedData startTime endTime**

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

pwminput→get_reportFrequency() YPwmInput
pwminput→reportFrequency()YPwmInput
get_reportFrequency

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

YPwmInput **target** **get_reportFrequency**

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

pwminput→get_resolution() YPwmInput pwminput→resolution()YPwmInput get_resolution

Returns the resolution of the measured values.

```
YPwmInput target get_resolution
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

pwminput→get_unit() YPwmInput
pwminput→unit()YPwmInput get_unit

Returns the measuring unit for the values returned by get_currentValue and callbacks.

```
YPwmInput target get_unit
```

That unit will change according to the pwmReportMode settings.

Returns :
a string corresponding to the measuring unit for the values returned by get_currentValue and callbacks

On failure, throws an exception or returns `Y_UNIT_INVALID`.

pwminput→loadCalibrationPoints()YPwmInput YPwmInput loadCalibrationPoints

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

YPwmInput **target loadCalibrationPoints rawValues refValues**

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwminput→set_highestValue() YPwmInput
pwminput→setHighestValue()YPwmInput
set_highestValue

Changes the recorded maximal value observed.

YPwmInput **target** **set_highestValue** **newval**

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwminput→set_logFrequency() YPwmInput
pwminput→setLogFrequency()YPwmInput
set_logFrequency

Changes the datalogger recording frequency for this function.

```
YPwmInput target set_logFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwminput→set_logicalName() YPwmInput
pwminput→setLogicalName()YPwmInput
set_logicalName

Changes the logical name of the voltage sensor.

```
YPwmInput 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 sensor.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwminput→set_lowestValue() YPwmInput
pwminput→setLowestValue()YPwmInput
set_lowestValue

Changes the recorded minimal value observed.

```
YPwmInput target set_lowestValue newval
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwminput→set_pwmReportMode() YPwmInput
pwminput→setPwmReportMode()YPwmInput
set_pwmReportMode

Modify the parameter type(frequency/duty cycle, pulse width ou edge count) returned by the get_currentValue function and callbacks.

YPwmInput **target** **set_pwmReportMode** **newval**

The edge count value will be limited to the 6 lowest digit, for values greater than one million, use get_pulseCounter().

Parameters :

newval a value among Y_PWMREPORTMODE_PWM_DUTYCYCLE, Y_PWMREPORTMODE_PWM_FREQUENCY, Y_PWMREPORTMODE_PWM_PULSEDURATION and Y_PWMREPORTMODE_PWM_EDGECOUNT

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwminput→set_reportFrequency() YPwmInput
pwminput→setReportFrequency()YPwmInput
set_reportFrequency

Changes the timed value notification frequency for this function.

```
YPwmInput target set_reportFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwminput→set_resolution() YPwmInput
pwminput→setResolution()YPwmInput
set_resolution

Changes the resolution of the measured physical values.

```
YPwmInput target set_resolution newval
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.33. Pwm function interface

The Yoctopuce application programming interface allows you to configure, start, and stop the PWM.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_pwmoutput.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YPwmOutput = yoctolib.YPwmOutput;
php     require_once('yocto_pwmoutput.php');
cpp     #include "yocto_pwmoutput.h"
m       #import "yocto_pwmoutput.h"
pas     uses yocto_pwmoutput;
vb      yocto_pwmoutput.vb
cs      yocto_pwmoutput.cs
java    import com.yoctopuce.YoctoAPI.YPwmOutput;
py      from yocto_pwmoutput import *
```

Global functions
yFindPwmOutput(**func**) Retrieves a PWM for a given identifier.
yFirstPwmOutput() Starts the enumeration of PWMs currently accessible.
`YPwmOutput` methods
pwmoutput→describe() Returns a short text that describes unambiguously the instance of the PWM in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
pwmoutput→dutyCycleMove(**target**, **ms_duration**) Performs a smooth change of the pulse duration toward a given value.
pwmoutput→get_advertisedValue() Returns the current value of the PWM (no more than 6 characters).
pwmoutput→get_dutyCycle() Returns the PWM duty cycle, in per cents.
pwmoutput→get_dutyCycleAtPowerOn() Returns the PWMs duty cycle at device power on as a floating point number between 0 and 100
pwmoutput→get_enabled() Returns the state of the PWMs.
pwmoutput→get_enabledAtPowerOn() Returns the state of the PWM at device power on.
pwmoutput→get_errorMessage() Returns the error message of the latest error with the PWM.
pwmoutput→get_errorType() Returns the numerical error code of the latest error with the PWM.
pwmoutput→get_frequency() Returns the PWM frequency in Hz.
pwmoutput→get_friendlyName() Returns a global identifier of the PWM in the format `MODULE_NAME.FUNCTION_NAME`.
pwmoutput→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

pwmoutput→get_functionId()
Returns the hardware identifier of the PWM, without reference to the module.

pwmoutput→get_hardwareId()
Returns the unique hardware identifier of the PWM in the form SERIAL.FUNCTIONID.

pwmoutput→get_logicalName()
Returns the logical name of the PWM.

pwmoutput→get_module()
Gets the YModule object for the device on which the function is located.

pwmoutput→get_module_async(**callback**, **context**)
Gets the YModule object for the device on which the function is located (asynchronous version).

pwmoutput→get_period()
Returns the PWM period in milliseconds.

pwmoutput→get_pulseDuration()
Returns the PWM pulse length in milliseconds, as a floating point number.

pwmoutput→get_userData()
Returns the value of the userData attribute, as previously stored using method set_userData.

pwmoutput→isOnline()
Checks if the PWM is currently reachable, without raising any error.

pwmoutput→isOnline_async(**callback**, **context**)
Checks if the PWM is currently reachable, without raising any error (asynchronous version).

pwmoutput→load(**msValidity**)
Preloads the PWM cache with a specified validity duration.

pwmoutput→load_async(**msValidity**, **callback**, **context**)
Preloads the PWM cache with a specified validity duration (asynchronous version).

pwmoutput→nextPwmOutput()
Continues the enumeration of PWMs started using yFirstPwmOutput().

pwmoutput→pulseDurationMove(**ms_target**, **ms_duration**)
Performs a smooth transistion of the pulse duration toward a given value.

pwmoutput→registerValueCallback(**callback**)
Registers the callback function that is invoked on every change of advertised value.

pwmoutput→set_dutyCycle(**newval**)
Changes the PWM duty cycle, in per cents.

pwmoutput→set_dutyCycleAtPowerOn(**newval**)
Changes the PWM duty cycle at device power on.

pwmoutput→set_enabled(**newval**)
Stops or starts the PWM.

pwmoutput→set_enabledAtPowerOn(**newval**)
Changes the state of the PWM at device power on.

pwmoutput→set_frequency(**newval**)
Changes the PWM frequency.

pwmoutput→set_logicalName(**newval**)
Changes the logical name of the PWM.

pwmoutput→set_period(**newval**)
Changes the PWM period in milliseconds.

pwmoutput→set_pulseDuration(newval)

Changes the PWM pulse length, in milliseconds.

pwmoutput→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

pwmoutput→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

pwmoutput→dutyCycleMove()YPwmOutput dutyCycleMove YPwmOutput

Performs a smooth change of the pulse duration toward a given value.

YPwmOutput **target dutyCycleMove target ms_duration**

Parameters :

target	new duty cycle at the end of the transition (floating-point number, between 0 and 1)
ms_duration	total duration of the transition, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→get_advertisedValue() YPwmOutput
pwmoutput→advertisedValue()YPwmOutput
get_advertisedValue

Returns the current value of the PWM (no more than 6 characters).

YPwmOutput **target** **get_advertisedValue**

Returns :
a string corresponding to the current value of the PWM (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

pwmoutput→get_dutyCycle() YPwmOutput
pwmoutput→dutyCycle()YPwmOutput get_dutyCycle

Returns the PWM duty cycle, in per cents.

```
YPwmOutput target get_dutyCycle
```

Returns :

a floating point number corresponding to the PWM duty cycle, in per cents

On failure, throws an exception or returns `Y_DUTYCYCLE_INVALID`.

pwmoutput→get_enabled() YPwmOutput pwmoutput→enabled()YPwmOutput get_enabled

Returns the state of the PWMs.

YPwmOutput **target** **get_enabled**

Returns :

either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to the state of the PWMs

On failure, throws an exception or returns Y_ENABLED_INVALID.

pwmoutput→get_enabledAtPowerOn() YPwmOutput
pwmoutput→enabledAtPowerOn()YPwmOutput
get_enabledAtPowerOn

Returns the state of the PWM at device power on.

```
YPwmOutput target get_enabledAtPowerOn
```

Returns :

either Y_ENABLEDATPOWERON_FALSE or Y_ENABLEDATPOWERON_TRUE, according to the state of the PWM at device power on

On failure, throws an exception or returns Y_ENABLEDATPOWERON_INVALID.

pwmoutput→get_frequency() YPwmOutput
pwmoutput→frequency()YPwmOutput get_frequency

Returns the PWM frequency in Hz.

YPwmOutput **target** **get_frequency**

Returns :
a floating point number corresponding to the PWM frequency in Hz

On failure, throws an exception or returns `Y_FREQUENCY_INVALID`.

pwmoutput→get_logicalName() YPwmOutput
pwmoutput→logicalName()YPwmOutput
get_logicalName

Returns the logical name of the PWM.

```
YPwmOutput target get_logicalName
```

Returns :
a string corresponding to the logical name of the PWM.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

pwmoutput→get_period() YPwmOutput
pwmoutput→period()YPwmOutput get_period

Returns the PWM period in milliseconds.

YPwmOutput **target** **get_period**

Returns :
a floating point number corresponding to the PWM period in milliseconds

On failure, throws an exception or returns `Y_PERIOD_INVALID`.

pwmoutput→get_pulseDuration() YPwmOutput pwmoutput→pulseDuration()YPwmOutput get_pulseDuration

Returns the PWM pulse length in milliseconds, as a floating point number.

YPwmOutput **target get_pulseDuration**

Returns :
a floating point number corresponding to the PWM pulse length in milliseconds, as a floating point number

On failure, throws an exception or returns `Y_PULSEDURATION_INVALID`.

pwmoutput→pulseDurationMove()YPwmOutput pulseDurationMove YPwmOutput

Performs a smooth transistion of the pulse duration toward a given value.

```
YPwmOutput target pulseDurationMove ms_target ms_duration
```

Any period, frequency, duty cycle or pulse width change will cancel any ongoing transition process.

Parameters :

ms_target	new pulse duration at the end of the transition (floating-point number, representing the pulse duration in milliseconds)
ms_duration	total duration of the transition, in milliseconds

Returns :

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_dutyCycle() YPwmOutput
pwmoutput→setDutyCycle()YPwmOutput
set_dutyCycle

Changes the PWM duty cycle, in per cents.

```
YPwmOutput target set_dutyCycle newval
```

Parameters :

newval a floating point number corresponding to the PWM duty cycle, in per cents

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_dutyCycleAtPowerOn() YPwmOutput
pwmoutput→setDutyCycleAtPowerOn()YPwmOutput
set_dutyCycleAtPowerOn

Changes the PWM duty cycle at device power on.

```
YPwmOutput target set_dutyCycleAtPowerOn newval
```

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval a floating point number corresponding to the PWM duty cycle at device power on

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_enabled() YPwmOutput
pwmoutput→setEnabled()YPwmOutput set_enabled

Stops or starts the PWM.

YPwmOutput **target set_enabled newval**

Parameters :

newval either Y_ENABLED_FALSE or Y_ENABLED_TRUE

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_enabledAtPowerOn() YPwmOutput
pwmoutput→setEnabledAtPowerOn()YPwmOutput
set_enabledAtPowerOn

Changes the state of the PWM at device power on.

YPwmOutput **target set_enabledAtPowerOn newval**

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval either `Y_ENABLEDATPOWERON_FALSE` or `Y_ENABLEDATPOWERON_TRUE`, according to the state of the PWM at device power on

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_frequency() YPwmOutput
pwmoutput→setFrequency()YPwmOutput
set_frequency

Changes the PWM frequency.

```
YPwmOutput target set_frequency newval
```

The duty cycle is kept unchanged thanks to an automatic pulse width change.

Parameters :

newval a floating point number corresponding to the PWM frequency

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_logicalName() YPwmOutput
pwmoutput→setLogicalName()YPwmOutput
set_logicalName

Changes the logical name of the PWM.

```
YPwmOutput 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 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()YPwmOutput set_period

Changes the PWM period in milliseconds.

YPwmOutput **target** **set_period** **newval**

Parameters :

newval a floating point number corresponding to the PWM period in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

pwmoutput→set_pulseDuration() YPwmOutput
pwmoutput→setPulseDuration()YPwmOutput
set_pulseDuration

Changes the PWM pulse length, in milliseconds.

```
YPwmOutput target set_pulseDuration newval
```

A pulse length cannot be longer than period, otherwise it is truncated.

Parameters :

newval a floating point number corresponding to the PWM pulse length, in milliseconds

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.34. PwmPowerSource function interface

The Yoctopuce application programming interface allows you to configure the voltage source used by all PWM on the same device.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_pwmpowersource.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YPwmPowerSource = yoctolib.YPwmPowerSource;
php     require_once('yocto_pwmpowersource.php');
cpp     #include "yocto_pwmpowersource.h"
m       #import "yocto_pwmpowersource.h"
pas     uses yocto_pwmpowersource;
vb      yocto_pwmpowersource.vb
cs      yocto_pwmpowersource.cs
java    import com.yoctopuce.YoctoAPI.YPwmPowerSource;
py      from yocto_pwmpowersource import *
```

Global functions
yFindPwmPowerSource(**func**) 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.

pwmpowersource→get_advertisedValue() YPwmPowerSource
pwmpowersource→advertisedValue()
YPwmPowerSource get_advertisedValue

Returns the current value of the voltage source (no more than 6 characters).

```
YPwmPowerSource 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`.

pwmpowersource→get_logicalName() YPwmPowerSource
pwmpowersource→logicalName()
YPwmPowerSource get_logicalName

Returns the logical name of the voltage source.

YPwmPowerSource **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`.

pwmpowersource→set_logicalName() YPwmPowerSource
pwmpowersource→setLogicalName()
YPwmPowerSource set_logicalName

Changes the logical name of the voltage source.

```
YPwmPowerSource 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.

pwmpowersource→set_powerMode() YPwmPowerSource
pwmpowersource→setPowerMode()
YPwmPowerSource set_powerMode

Changes the PWM power source.

YPwmPowerSource **target set_powerMode newval**

PWM can use isolated 5V from USB, isolated 3V from USB or voltage from an external power source. The PWM can also work in open drain mode. In that mode, the PWM actively pulls the line down. Warning: this setting is common to all PWM on the same device. If you change that parameter, all PWM located on the same device are affected. If you want the change to be kept after a device reboot, make sure to call the matching module `saveToFlash()`.

Parameters :

newval a value among `Y_POWERMODE_USB_5V`, `Y_POWERMODE_USB_3V`, `Y_POWERMODE_EXT_V` and `Y_POWERMODE_OPNDRN` corresponding to the PWM power source

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.35. Quaternion interface

The Yoctopuce API YQt class provides direct access to the Yocto3D attitude estimation using a quaternion. It is usually not needed to use the YQt class directly, as the YGyro class provides a more convenient higher-level interface.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_gyro.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YGyro = yoctolib.YGyro;
php     require_once('yocto_gyro.php');
cpp     #include "yocto_gyro.h"
m       #import "yocto_gyro.h"
pas     uses yocto_gyro;
vb      yocto_gyro.vb
cs      yocto_gyro.cs
java    import com.yoctopuce.YoctoAPI.YGyro;
py      from yocto_gyro import *
```

Global functions
yFindQt(**func**) Retrieves a quaternion component for a given identifier.
yFirstQt() Starts the enumeration of quaternion components currently accessible.
YQt methods
qt→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
qt→describe() Returns a short text that describes unambiguously the instance of the quaternion component in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
qt→get_advertisedValue() Returns the current value of the quaternion component (no more than 6 characters).
qt→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in units, as a floating point number.
qt→get_currentValue() Returns the current value of the value, in units, as a floating point number.
qt→get_errorMessage() Returns the error message of the latest error with the quaternion component.
qt→get_errorType() Returns the numerical error code of the latest error with the quaternion component.
qt→get_friendlyName() Returns a global identifier of the quaternion component in the format `MODULE_NAME.FUNCTION_NAME`.
qt→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
qt→get_functionId() Returns the hardware identifier of the quaternion component, without reference to the module.
qt→get_hardwareId()

Returns the unique hardware identifier of the quaternion component in the form `SERIAL.FUNCTIONID`.

qt→get_highestValue()

Returns the maximal value observed for the value since the device was started.

qt→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

qt→get_logicalName()

Returns the logical name of the quaternion component.

qt→get_lowestValue()

Returns the minimal value observed for the value since the device was started.

qt→get_module()

Gets the `YModule` object for the device on which the function is located.

qt→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

qt→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

qt→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

qt→get_resolution()

Returns the resolution of the measured values.

qt→get_unit()

Returns the measuring unit for the value.

qt→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

qt→isOnline()

Checks if the quaternion component is currently reachable, without raising any error.

qt→isOnline_async(**callback**, **context**)

Checks if the quaternion component is currently reachable, without raising any error (asynchronous version).

qt→load(**msValidity**)

Preloads the quaternion component cache with a specified validity duration.

qt→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

qt→load_async(**msValidity**, **callback**, **context**)

Preloads the quaternion component cache with a specified validity duration (asynchronous version).

qt→nextQt()

Continues the enumeration of quaternion components started using `yFirstQt()`.

qt→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

qt→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

qt→set_highestValue(**newval**)

Changes the recorded maximal value observed.

qt→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

qt→set_logicalName(**newval**)

Changes the logical name of the quaternion component.

qt→set_lowestValue(**newval**)

Changes the recorded minimal value observed.

qt→set_reportFrequency(**newval**)

Changes the timed value notification frequency for this function.

qt→set_resolution(**newval**)

Changes the resolution of the measured physical values.

qt→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

qt→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

qt→calibrateFromPoints()YSensor calibrateFromPoints YQt

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

YSensor **target calibrateFromPoints rawValues refValues**

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→get_advertisedValue() YQt
qt→advertisedValue()YSensor get_advertisedValue

Returns the current value of the quaternion component (no more than 6 characters).

YSensor **target get_advertisedValue**

Returns :

a string corresponding to the current value of the quaternion component (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

qt→get_currentRawValue() YQt
qt→currentRawValue()YSensor get_currentRawValue

Returns the uncalibrated, unrounded raw value returned by the sensor, in units, as a floating point number.

YSensor **target get_currentRawValue**

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in units, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

qt→get_currentValue() YQt
qt→currentValue()YSensor get_currentValue

Returns the current value of the value, in units, as a floating point number.

YSensor **target get_currentValue**

Returns :
a floating point number corresponding to the current value of the value, in units, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

qt→get_highestValue() YQt
qt→highestValue()YSensor get_highestValue

Returns the maximal value observed for the value since the device was started.

YSensor **target get_highestValue**

Returns :

a floating point number corresponding to the maximal value observed for the value since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

qt→get_logFrequency() YQt
qt→logFrequency()YSensor get_logFrequency

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

YSensor **target get_logFrequency**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

qt→get_logicalName() YQt
qt→logicalName()YSensor get_logicalName

Returns the logical name of the quaternion component.

```
YSensor target get_logicalName
```

Returns :

a string corresponding to the logical name of the quaternion component.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

qt→get_lowestValue() YQt
qt→lowestValue()YSensor get_lowestValue

Returns the minimal value observed for the value since the device was started.

YSensor **target get_lowestValue**

Returns :

a floating point number corresponding to the minimal value observed for the value since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

qt→get_recordedData() YQt
qt→recordedData()YSensor get_recordedData

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YSensor **target get_recordedData startTime endTime**

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

qt→get_reportFrequency() YQt
qt→reportFrequency()YSensor get_reportFrequency

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

YSensor **target get_reportFrequency**

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

qt→get_resolution() YQt
qt→resolution()YSensor get_resolution

Returns the resolution of the measured values.

```
YSensor target get_resolution
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

qt→get_unit() YQt
qt→unit()YSensor get_unit

Returns the measuring unit for the value.

YSensor **target** **get_unit**

Returns :
a string corresponding to the measuring unit for the value

On failure, throws an exception or returns `Y_UNIT_INVALID`.

qt→loadCalibrationPoints()YSensor loadCalibrationPoints YQt

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

YSensor **target loadCalibrationPoints rawValues refValues**

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_highestValue() YQt
qt→setHighestValue()YSensor set_highestValue

Changes the recorded maximal value observed.

```
YSensor target set_highestValue newval
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_logFrequency() YQt
qt→setLogFrequency()YSensor set_logFrequency

Changes the datalogger recording frequency for this function.

```
YSensor target set_logFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_logicalName() YQt
qt→setLogicalName()YSensor set_logicalName

Changes the logical name of the quaternion component.

```
YSensor 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 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()YSensor set_lowestValue

Changes the recorded minimal value observed.

YSensor **target set_lowestValue newval**

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_reportFrequency() YQt
qt→setReportFrequency()YSensor
set_reportFrequency

Changes the timed value notification frequency for this function.

```
YSensor target set_reportFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

qt→set_resolution() YQt
qt→setResolution()YSensor set_resolution

Changes the resolution of the measured physical values.

```
YSensor target set_resolution newval
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.36. Real Time Clock function interface

The RealTimeClock function maintains and provides current date and time, even accross power cut lasting several days. It is the base for automated wake-up functions provided by the WakeUpScheduler. The current time may represent a local time as well as an UTC time, but no automatic time change will occur to account for daylight saving time.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_realtimeclock.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YRealTimeClock = yoctolib.YRealTimeClock;
php     require_once('yocto_realtimeclock.php');
cpp     #include "yocto_realtimeclock.h"
m       #import "yocto_realtimeclock.h"
pas     uses yocto_realtimeclock;
vb      yocto_realtimeclock.vb
cs      yocto_realtimeclock.cs
java    import com.yoctopuce.YoctoAPI.YRealTimeClock;
py      from yocto_realtimeclock import *
```

Global functions
yFindRealTimeClock(**func**) Retrieves a clock for a given identifier.
yFirstRealTimeClock() Starts the enumeration of clocks currently accessible.
`YRealTimeClock` methods
realtimeclock→describe() Returns a short text that describes unambiguously the instance of the clock in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
realtimeclock→get_advertisedValue() Returns the current value of the clock (no more than 6 characters).
realtimeclock→get_dateTime() Returns the current time in the form "YYYY/MM/DD hh:mm:ss"
realtimeclock→get_errorMessage() Returns the error message of the latest error with the clock.
realtimeclock→get_errorType() Returns the numerical error code of the latest error with the clock.
realtimeclock→get_friendlyName() Returns a global identifier of the clock in the format `MODULE_NAME.FUNCTION_NAME`.
realtimeclock→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
realtimeclock→get_functionId() Returns the hardware identifier of the clock, without reference to the module.
realtimeclock→get_hardwareId() Returns the unique hardware identifier of the clock in the form `SERIAL.FUNCTIONID`.
realtimeclock→get_logicalName() Returns the logical name of the clock.
realtimeclock→get_module()

Gets the YModule object for the device on which the function is located.

realtimeclock→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

realtimeclock→get_timeSet()

Returns true if the clock has been set, and false otherwise.

realtimeclock→get_unixTime()

Returns the current time in Unix format (number of elapsed seconds since Jan 1st, 1970).

realtimeclock→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

realtimeclock→get_utcOffset()

Returns the number of seconds between current time and UTC time (time zone).

realtimeclock→isOnline()

Checks if the clock is currently reachable, without raising any error.

realtimeclock→isOnline_async(callback, context)

Checks if the clock is currently reachable, without raising any error (asynchronous version).

realtimeclock→load(msValidity)

Preloads the clock cache with a specified validity duration.

realtimeclock→load_async(msValidity, callback, context)

Preloads the clock cache with a specified validity duration (asynchronous version).

realtimeclock→nextRealTimeClock()

Continues the enumeration of clocks started using yFirstRealTimeClock().

realtimeclock→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

realtimeclock→set_logicalName(newval)

Changes the logical name of the clock.

realtimeclock→set_unixTime(newval)

Changes the current time.

realtimeclock→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

realtimeclock→set_utcOffset(newval)

Changes the number of seconds between current time and UTC time (time zone).

realtimeclock→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

realtimeclock→get_advertisedValue() YRealTimeClock
realtimeclock→advertisedValue()YRealTimeClock
get_advertisedValue

Returns the current value of the clock (no more than 6 characters).

YRealTimeClock **target get_advertisedValue**

Returns :
a string corresponding to the current value of the clock (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

realtimeclock→get_logicalName() YRealTimeClock
realtimeclock→logicalName()YRealTimeClock
get_logicalName

Returns the logical name of the clock.

YRealTimeClock **target get_logicalName**

Returns :
a string corresponding to the logical name of the clock.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

realtimeclock→get_timeSet() YRealTimeClock
realtimeclock→timeSet()YRealTimeClock
get_timeSet

Returns true if the clock has been set, and false otherwise.

YRealTimeClock **target get_timeSet**

Returns :

either `Y_TIMESET_FALSE` or `Y_TIMESET_TRUE`, according to true if the clock has been set, and false otherwise

On failure, throws an exception or returns `Y_TIMESET_INVALID`.

realtimeclock→get_unixTime() YRealTimeClock
realtimeclock→unixTime()YRealTimeClock
get_unixTime

Returns the current time in Unix format (number of elapsed seconds since Jan 1st, 1970).

YRealTimeClock **target get_unixTime**

Returns :
an integer corresponding to the current time in Unix format (number of elapsed seconds since Jan 1st, 1970)

On failure, throws an exception or returns `Y_UNIXTIME_INVALID`.

realtimeclock→get_utcOffset() YRealTimeClock
realtimeclock→utcOffset()YRealTimeClock
get_utcOffset

Returns the number of seconds between current time and UTC time (time zone).

YRealTimeClock **target get_utcOffset**

Returns :
an integer corresponding to the number of seconds between current time and UTC time (time zone)

On failure, throws an exception or returns `Y_UTCOFFSET_INVALID`.

realtimeclock→set_logicalName() YRealTimeClock
realtimeclock→setLogicalName()YRealTimeClock
set_logicalName

Changes the logical name of the clock.

```
YRealTimeClock 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 clock.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

realtimeclock→set_unixTime() YRealTimeClock
realtimeclock→setUnixTime()YRealTimeClock
set_unixTime

Changes the current time.

```
YRealTimeClock target set_unixTime newval
```

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_utcOffset() YRealTimeClock
realtimeclock→setUtcOffset()YRealTimeClock
set_utcOffset

Changes the number of seconds between current time and UTC time (time zone).

```
YRealTimeClock target set_utcOffset newval
```

The timezone is automatically rounded to the nearest multiple of 15 minutes. If current UTC time is known, the current time will automatically be updated according to the selected time zone.

Parameters :

newval an integer corresponding to the number of seconds between current time and UTC time (time zone)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.37. Reference frame configuration

This class is used to setup the base orientation of the Yocto-3D, so that the orientation functions, relative to the earth surface plane, use the proper reference frame. The class also implements a tridimensional sensor calibration process, which can compensate for local variations of standard gravity and improve the precision of the tilt sensors.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_refframe.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YRefFrame = yoctolib.YRefFrame;
php     require_once('yocto_refframe.php');
cpp     #include "yocto_refframe.h"
m       #import "yocto_refframe.h"
pas     uses yocto_refframe;
vb      yocto_refframe.vb
cs      yocto_refframe.cs
java    import com.yoctopuce.YoctoAPI.YRefFrame;
py      from yocto_refframe import *
```

Global functions
yFindRefFrame(**func**) Retrieves a reference frame for a given identifier.
yFirstRefFrame() Starts the enumeration of reference frames currently accessible.
`YRefFrame` methods
refframe→cancel3DCalibration() Aborts the sensors tridimensional calibration process et restores normal settings.
refframe→describe() Returns a short text that describes unambiguously the instance of the reference frame in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
refframe→get_3DCalibrationHint() Returns instructions to proceed to the tridimensional calibration initiated with method `start3DCalibration`.
refframe→get_3DCalibrationLogMsg() Returns the latest log message from the calibration process.
refframe→get_3DCalibrationProgress() Returns the global process indicator for the tridimensional calibration initiated with method `start3DCalibration`.
refframe→get_3DCalibrationStage() Returns index of the current stage of the calibration initiated with method `start3DCalibration`.
refframe→get_3DCalibrationStageProgress() Returns the process indicator for the current stage of the calibration initiated with method `start3DCalibration`.
refframe→get_advertisedValue() Returns the current value of the reference frame (no more than 6 characters).
refframe→get_bearing() Returns the reference bearing used by the compass.

reframe→get_errorMessage()

Returns the error message of the latest error with the reference frame.

reframe→get_errorType()

Returns the numerical error code of the latest error with the reference frame.

reframe→get_friendlyName()

Returns a global identifier of the reference frame in the format `MODULE_NAME.FUNCTION_NAME`.

reframe→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

reframe→get_functionId()

Returns the hardware identifier of the reference frame, without reference to the module.

reframe→get_hardwareId()

Returns the unique hardware identifier of the reference frame in the form `SERIAL.FUNCTIONID`.

reframe→get_logicalName()

Returns the logical name of the reference frame.

reframe→get_module()

Gets the `YModule` object for the device on which the function is located.

reframe→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

reframe→get_mountOrientation()

Returns the installation orientation of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

reframe→get_mountPosition()

Returns the installation position of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

reframe→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

reframe→isOnline()

Checks if the reference frame is currently reachable, without raising any error.

reframe→isOnline_async(**callback**, **context**)

Checks if the reference frame is currently reachable, without raising any error (asynchronous version).

reframe→load(**msValidity**)

Preloads the reference frame cache with a specified validity duration.

reframe→load_async(**msValidity**, **callback**, **context**)

Preloads the reference frame cache with a specified validity duration (asynchronous version).

reframe→more3DCalibration()

Continues the sensors tridimensional calibration process previously initiated using method `start3DCalibration`.

reframe→nextRefFrame()

Continues the enumeration of reference frames started using `yFirstRefFrame()`.

reframe→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

reframe→save3DCalibration()

Applies the sensors tridimensional calibration parameters that have just been computed.

reframe→set_bearing(**newval**)

Changes the reference bearing used by the compass.

reframe→set_logicalName(**newval**)

Changes the logical name of the reference frame.

refframe→set_mountPosition(position, orientation)

Changes the compass and tilt sensor frame of reference.

refframe→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

refframe→start3DCalibration()

Initiates the sensors tridimensional calibration process.

refframe→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

refframe→cancel3DCalibration()YRefFrame cancel3DCalibration

YRefFrame

Aborts the sensors tridimensional calibration process et restores normal settings.

```
YRefFrame target cancel3DCalibration
```

On failure, throws an exception or returns a negative error code.

refframe→get_3DCalibrationHint() YRefFrame
refframe→3DCalibrationHint()YRefFrame
get_3DCalibrationHint

Returns instructions to proceed to the tridimensional calibration initiated with method `start3DCalibration`.

YRefFrame **target** **get_3DCalibrationHint**

Returns :
a character string.

refframe→get_3DCalibrationLogMsg() YRefFrame
refframe→3DCalibrationLogMsg()YRefFrame
get_3DCalibrationLogMsg

Returns the latest log message from the calibration process.

```
YRefFrame target get_3DCalibrationLogMsg
```

When no new message is available, returns an empty string.

Returns :
a character string.

refframe→get_3DCalibrationProgress() YRefFrame
refframe→3DCalibrationProgress()YRefFrame
get_3DCalibrationProgress

Returns the global process indicator for the tridimensional calibration initiated with method `start3DCalibration`.

YRefFrame **target** **get_3DCalibrationProgress**

Returns :
an integer between 0 (not started) and 100 (stage completed).

refframe→get_3DCalibrationStage() YRefFrame
refframe→3DCalibrationStage()YRefFrame
get_3DCalibrationStage

Returns index of the current stage of the calibration initiated with method `start3DCalibration`.

YRefFrame **target get_3DCalibrationStage**

Returns :
an integer, growing each time a calibration stage is completed.

refframe→get_3DCalibrationStageProgress() YRefFrame
refframe→3DCalibrationStageProgress()YRefFrame
get_3DCalibrationStageProgress

Returns the process indicator for the current stage of the calibration initiated with method `start3DCalibration`.

YRefFrame **target** **get_3DCalibrationStageProgress**

Returns :
an integer between 0 (not started) and 100 (stage completed).

refframe→get_advertisedValue() YRefFrame
refframe→advertisedValue()YRefFrame
get_advertisedValue

Returns the current value of the reference frame (no more than 6 characters).

```
YRefFrame target get_advertisedValue
```

Returns :
a string corresponding to the current value of the reference frame (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

refframe→get_bearing() YRefFrame
refframe→bearing()YRefFrame get_bearing

Returns the reference bearing used by the compass.

```
YRefFrame target get_bearing
```

The relative bearing indicated by the compass is the difference between the measured magnetic heading and the reference bearing indicated here.

Returns :
a floating point number corresponding to the reference bearing used by the compass

On failure, throws an exception or returns `Y_BEARING_INVALID`.

refframe→get_logicalName() YRefFrame
refframe→logicalName()YRefFrame get_logicalName

Returns the logical name of the reference frame.

YRefFrame **target get_logicalName**

Returns :
a string corresponding to the logical name of the reference frame.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

refframe→get_mountOrientation() YRefFrame
refframe→mountOrientation()YRefFrame
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.

YRefFrame **target** **get_mountOrientation**

Returns :

a value among the enumeration Y_MOUNTORIENTATION (Y_MOUNTORIENTATION_TWELVE, Y_MOUNTORIENTATION_THREE, Y_MOUNTORIENTATION_SIX, Y_MOUNTORIENTATION_NINE) corresponding to the orientation of the "X" arrow on the device, as on a clock dial seen from an observer in the center of the box. On the bottom face, the 12H orientation points to the front, while on the top face, the 12H orientation points to the rear.

On failure, throws an exception or returns a negative error code.

refframe→get_mountPosition() YRefFrame
refframe→mountPosition()YRefFrame
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.

YRefFrame **target get_mountPosition**

Returns :

a value among the `Y_MOUNTPOSITION` enumeration (`Y_MOUNTPOSITION_BOTTOM`, `Y_MOUNTPOSITION_TOP`, `Y_MOUNTPOSITION_FRONT`, `Y_MOUNTPOSITION_RIGHT`, `Y_MOUNTPOSITION_REAR`, `Y_MOUNTPOSITION_LEFT`), corresponding to the installation in a box, on one of the six faces.

On failure, throws an exception or returns a negative error code.

refframe→more3DCalibration()YRefFrame YRefFrame more3DCalibration

Continues the sensors tridimensional calibration process previously initiated using method `start3DCalibration`.

YRefFrame **target** **more3DCalibration**

This method should be called approximately 5 times per second, while positioning the device according to the instructions provided by method `get_3DCalibrationHint`. Note that the instructions change during the calibration process. On failure, throws an exception or returns a negative error code.

refframe→save3DCalibration()YRefFrame save3DCalibration

YRefFrame

Applies the sensors tridimensional calibration parameters that have just been computed.

YRefFrame **target save3DCalibration**

Remember to call the `saveToFlash()` method of the module if the changes must be kept when the device is restarted. On failure, throws an exception or returns a negative error code.

refframe→set_bearing() YRefFrame
refframe→setBearing()YRefFrame set_bearing

Changes the reference bearing used by the compass.

```
YRefFrame target set_bearing newval
```

The relative bearing indicated by the compass is the difference between the measured magnetic heading and the reference bearing indicated here. For instance, if you setup as reference bearing the value of the earth magnetic declination, the compass will provide the orientation relative to the geographic North. Similarly, when the sensor is not mounted along the standard directions because it has an additional yaw angle, you can set this angle in the reference bearing so that the compass provides the expected natural direction. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a floating point number corresponding to the reference bearing used by the compass

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

refframe→set_logicalName() YRefFrame
refframe→setLogicalName()YRefFrame
set_logicalName

Changes the logical name of the reference frame.

```
YRefFrame 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 reference frame.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

refframe→set_mountPosition() YRefFrame
refframe→setMountPosition()YRefFrame
set_mountPosition

Changes the compass and tilt sensor frame of reference.

YRefFrame **target set_mountPosition position orientation**

The magnetic compass and the tilt sensors (pitch and roll) naturally work in the plane parallel to the earth surface. In case the device is not installed upright and horizontally, you must select its reference orientation (parallel to the earth surface) so that the measures are made relative to this position.

Parameters :

position a value among the `Y_MOUNTPOSITION` enumeration (`Y_MOUNTPOSITION_BOTTOM`, `Y_MOUNTPOSITION_TOP`, `Y_MOUNTPOSITION_FRONT`, `Y_MOUNTPOSITION_RIGHT`, `Y_MOUNTPOSITION_REAR`, `Y_MOUNTPOSITION_LEFT`), corresponding to the installation in a box, on one of the six faces.

orientation a value among the enumeration `Y_MOUNTORIENTATION` (`Y_MOUNTORIENTATION_TWELVE`, `Y_MOUNTORIENTATION_THREE`, `Y_MOUNTORIENTATION_SIX`, `Y_MOUNTORIENTATION_NINE`) corresponding to the orientation of the "X" arrow on the device, as on a clock dial seen from an observer in the center of the box. On the bottom face, the 12H orientation points to the front, while on the top face, the 12H orientation points to the rear. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

refframe→start3DCalibration()YRefFrame start3DCalibration YRefFrame

Initiates the sensors tridimensional calibration process.

YRefFrame **target start3DCalibration**

This calibration is used at low level for inertial position estimation and to enhance the precision of the tilt sensors. After calling this method, the device should be moved according to the instructions provided by method `get_3DCalibrationHint`, and `more3DCalibration` should be invoked about 5 times per second. The calibration procedure is completed when the method `get_3DCalibrationProgress` returns 100. At this point, the computed calibration parameters can be applied using method `save3DCalibration`. The calibration process can be canceled at any time using method `cancel3DCalibration`. On failure, throws an exception or returns a negative error code.

3.38. Relay function interface

The Yoctopuce application programming interface allows you to switch the relay state. This change is not persistent: the relay will automatically return to its idle position whenever power is lost or if the module is restarted. The library can also generate automatically short pulses of determined duration. On devices with two output for each relay (double throw), the two outputs are named A and B, with output A corresponding to the idle position (at power off) and the output B corresponding to the active state. If you prefer the alternate default state, simply switch your cables on the board.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_relay.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YRelay = yoctolib.YRelay;
php     require_once('yocto_relay.php');
cpp     #include "yocto_relay.h"
m       #import "yocto_relay.h"
pas     uses yocto_relay;
vb      yocto_relay.vb
cs      yocto_relay.cs
java    import com.yoctopuce.YoctoAPI.YRelay;
py      from yocto_relay import *
```

Global functions
yFindRelay(**func**) 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**)

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.

relay→delayedPulse()YRelay delayedPulse YRelay

Schedules a pulse.

YRelay **target delayedPulse ms_delay ms_duration**

Parameters :

ms_delay	waiting time before the pulse, in millisecondes
ms_duration	pulse duration, in millisecondes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→get_advertisedValue() YRelay
relay→advertisedValue()YRelay get_advertisedValue

Returns the current value of the relay (no more than 6 characters).

YRelay **target** **get_advertisedValue**

Returns :

a string corresponding to the current value of the relay (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

relay→get_countdown() YRelay relay→countdown()YRelay get_countdown

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

YRelay **target** **get_countdown**

Returns :

an integer corresponding to the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero

On failure, throws an exception or returns `Y_COUNTDOWN_INVALID`.

relay→get_logicalName() YRelay
relay→logicalName()YRelay get_logicalName

Returns the logical name of the relay.

YRelay **target** **get_logicalName**

Returns :
a string corresponding to the logical name of the relay.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

relay→get_maxTimeOnStateA() YRelay
relay→maxTimeOnStateA()YRelay
get_maxTimeOnStateA

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

YRelay **target** **get_maxTimeOnStateA**

Zero means no maximum time.

Returns :
an integer

On failure, throws an exception or returns `Y_MAXTIMEONSTATEA_INVALID`.

relay→get_maxTimeOnStateB() YRelay relay→maxTimeOnStateB()YRelay get_maxTimeOnStateB

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

YRelay **target** **get_maxTimeOnStateB**

Zero means no maximum time.

Returns :
an integer

On failure, throws an exception or returns `Y_MAXTIMEONSTATEB_INVALID`.

relay→get_output() YRelay relay→output()YRelay get_output

Returns the output state of the relays, when used as a simple switch (single throw).

```
YRelay target get_output
```

Returns :

either `Y_OUTPUT_OFF` or `Y_OUTPUT_ON`, according to the output state of the relays, when used as a simple switch (single throw)

On failure, throws an exception or returns `Y_OUTPUT_INVALID`.

relay→get_pulseTimer() YRelay relay→pulseTimer()YRelay get_pulseTimer

Returns the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation.

```
YRelay target get_pulseTimer
```

When there is no ongoing pulse, returns zero.

Returns :

an integer corresponding to the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation

On failure, throws an exception or returns `Y_PULSETIMER_INVALID`.

relay→get_state() YRelay
relay→state()YRelay get_state

Returns the state of the relays (A for the idle position, B for the active position).

YRelay **target** **get_state**

Returns :

either Y_STATE_A or Y_STATE_B, according to the state of the relays (A for the idle position, B for the active position)

On failure, throws an exception or returns Y_STATE_INVALID.

relay→get_stateAtPowerOn() YRelay relay→stateAtPowerOn()YRelay 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).

YRelay **target get_stateAtPowerOn**

Returns :

a value among `Y_STATEATPOWERON_UNCHANGED`, `Y_STATEATPOWERON_A` and `Y_STATEATPOWERON_B` corresponding to the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED for no change)

On failure, throws an exception or returns `Y_STATEATPOWERON_INVALID`.

relay→pulse()YRelay pulse YRelay

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

YRelay **target** **pulse** **ms_duration**

Parameters :

ms_duration pulse duration, in millisecondes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_logicalName() YRelay relay→setLogicalName()YRelay set_logicalName

Changes the logical name of the relay.

```
YRelay 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 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()YRelay
set_maxTimeOnStateA

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

```
YRelay target set_maxTimeOnStateA newval
```

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_maxTimeOnStateB() YRelay relay→setMaxTimeOnStateB()YRelay set_maxTimeOnStateB

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

```
YRelay target set_maxTimeOnStateB newval
```

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_output() YRelay
relay→setOutput()YRelay set_output

Changes the output state of the relays, when used as a simple switch (single throw).

YRelay **target** **set_output** **newval**

Parameters :

newval either Y_OUTPUT_OFF or Y_OUTPUT_ON, according to the output state of the relays, when used as a simple switch (single throw)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_state() YRelay
relay→setState()YRelay set_state

Changes the state of the relays (A for the idle position, B for the active position).

YRelay **target** **set_state** **newval**

Parameters :

newval either Y_STATE_A or Y_STATE_B, according to the state of the relays (A for the idle position, B for the active position)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

relay→set_stateAtPowerOn() YRelay
relay→setStateAtPowerOn()YRelay
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).

YRelay **target set_stateAtPowerOn newval**

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval a value among `Y_STATEATPOWERON_UNCHANGED`, `Y_STATEATPOWERON_A` and `Y_STATEATPOWERON_B`

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.39. Sensor function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_api.js'></script>
nodejs  var yoctolib = require('yoctolib');
        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, in the specified unit, as a floating point number.
sensor→get_currentValue() Returns the current value of the measure, in the specified unit, as a floating point number.
sensor→get_errorMessage() Returns the error message of the latest error with the sensor.
sensor→get_errorType() Returns the numerical error code of the latest error with the sensor.
sensor→get_friendlyName() Returns a global identifier of the sensor in the format `MODULE_NAME.FUNCTION_NAME`.
sensor→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
sensor→get_functionId() Returns the hardware identifier of the sensor, without reference to the module.
sensor→get_hardwareId()

Returns the unique hardware identifier of the sensor in the form `SERIAL.FUNCTIONID`.

sensor→get_highestValue()

Returns the maximal value observed for the measure since the device was started.

sensor→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

sensor→get_logicalName()

Returns the logical name of the sensor.

sensor→get_lowestValue()

Returns the minimal value observed for the measure since the device was started.

sensor→get_module()

Gets the `YModule` object for the device on which the function is located.

sensor→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

sensor→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

sensor→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

sensor→get_resolution()

Returns the resolution of the measured values.

sensor→get_unit()

Returns the measuring unit for the measure.

sensor→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

sensor→isOnline()

Checks if the sensor is currently reachable, without raising any error.

sensor→isOnline_async(**callback**, **context**)

Checks if the sensor is currently reachable, without raising any error (asynchronous version).

sensor→load(**msValidity**)

Preloads the sensor cache with a specified validity duration.

sensor→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

sensor→load_async(**msValidity**, **callback**, **context**)

Preloads the sensor cache with a specified validity duration (asynchronous version).

sensor→nextSensor()

Continues the enumeration of sensors started using `yFirstSensor()`.

sensor→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

sensor→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

sensor→set_highestValue(**newval**)

Changes the recorded maximal value observed.

sensor→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

sensor→set_logicalName(**newval**)

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.

sensor→calibrateFromPoints()YSensor calibrateFromPoints — YSensor

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

YSensor **target calibrateFromPoints rawValues refValues**

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→get_advertisedValue() YSensor
sensor→advertisedValue()YSensor
get_advertisedValue

Returns the current value of the sensor (no more than 6 characters).

YSensor **target** **get_advertisedValue**

Returns :
a string corresponding to the current value of the sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

sensor→get_currentRawValue() YSensor
sensor→currentRawValue()YSensor
get_currentRawValue

Returns the uncalibrated, unrounded raw value returned by the sensor, in the specified unit, as a floating point number.

YSensor **target get_currentRawValue**

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in the specified unit, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

sensor→get_currentValue() YSensor
sensor→currentValue()YSensor get_currentValue

Returns the current value of the measure, in the specified unit, as a floating point number.

YSensor **target** **get_currentValue**

Returns :

a floating point number corresponding to the current value of the measure, in the specified unit, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

sensor→get_highestValue() YSensor
sensor→highestValue()YSensor get_highestValue

Returns the maximal value observed for the measure since the device was started.

YSensor **target get_highestValue**

Returns :

a floating point number corresponding to the maximal value observed for the measure since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

sensor→get_logFrequency() YSensor
sensor→logFrequency()YSensor get_logFrequency

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

YSensor **target get_logFrequency**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

sensor→get_logicalName() YSensor
sensor→logicalName()YSensor get_logicalName

Returns the logical name of the sensor.

YSensor **target get_logicalName**

Returns :
a string corresponding to the logical name of the sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

sensor→get_lowestValue() YSensor
sensor→lowestValue()YSensor get_lowestValue

Returns the minimal value observed for the measure since the device was started.

YSensor **target** **get_lowestValue**

Returns :

a floating point number corresponding to the minimal value observed for the measure since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

sensor→get_recordedData() YSensor
sensor→recordedData()YSensor get_recordedData

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
YSensor target get_recordedData startTime endTime
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

sensor→get_reportFrequency() YSensor sensor→reportFrequency()YSensor get_reportFrequency

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
YSensor target get_reportFrequency
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

sensor→get_resolution() YSensor sensor→resolution()YSensor get_resolution

Returns the resolution of the measured values.

```
YSensor target get_resolution
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

sensor→get_unit() YSensor
sensor→unit()YSensor get_unit

Returns the measuring unit for the measure.

```
YSensor target get_unit
```

Returns :
a string corresponding to the measuring unit for the measure

On failure, throws an exception or returns `Y_UNIT_INVALID`.

sensor→loadCalibrationPoints()YSensor loadCalibrationPoints YSensor

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

YSensor **target loadCalibrationPoints rawValues refValues**

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→set_highestValue() YSensor sensor→setHighestValue()YSensor set_highestValue

Changes the recorded maximal value observed.

```
YSensor target set_highestValue newval
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→set_logFrequency() YSensor
sensor→setLogFrequency()YSensor
set_logFrequency

Changes the datalogger recording frequency for this function.

```
YSensor target set_logFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→set_logicalName() YSensor
sensor→setLogicalName()YSensor set_logicalName

Changes the logical name of the sensor.

```
YSensor 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 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()YSensor set_lowestValue

Changes the recorded minimal value observed.

```
YSensor target set_lowestValue newval
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→set_reportFrequency() YSensor
sensor→setReportFrequency()YSensor
set_reportFrequency

Changes the timed value notification frequency for this function.

YSensor **target set_reportFrequency newval**

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

sensor→set_resolution() YSensor sensor→setResolution()YSensor set_resolution

Changes the resolution of the measured physical values.

```
YSensor target set_resolution newval
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.40. SerialPort function interface

The SerialPort function interface allows you to fully drive a Yoctopuce serial port, to send and receive data, and to configure communication parameters (baud rate, bit count, parity, flow control and protocol). Note that Yoctopuce serial ports are not exposed as virtual COM ports. They are meant to be used in the same way as all Yoctopuce devices.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_serialport.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YSerialPort = yoctolib.YSerialPort;
php     require_once('yocto_serialport.php');
cpp     #include "yocto_serialport.h"
m       #import "yocto_serialport.h"
pas     uses yocto_serialport;
vb      yocto_serialport.vb
cs      yocto_serialport.cs
java    import com.yoctopuce.YoctoAPI.YSerialPort;
py      from yocto_serialport import *
```

Global functions
yFindSerialPort(**func**) Retrieves a serial port for a given identifier.
yFirstSerialPort() Starts the enumeration of serial ports currently accessible.
`YSerialPort` methods
serialport→describe() Returns a short text that describes unambiguously the instance of the serial port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
serialport→get_CTS() Read the level of the CTS line.
serialport→get_advertisedValue() Returns the current value of the serial port (no more than 6 characters).
serialport→get_errCount() Returns the total number of communication errors detected since last reset.
serialport→get_errorMessage() Returns the error message of the latest error with the serial port.
serialport→get_errorType() Returns the numerical error code of the latest error with the serial port.
serialport→get_friendlyName() Returns a global identifier of the serial port in the format `MODULE_NAME.FUNCTION_NAME`.
serialport→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
serialport→get_functionId() Returns the hardware identifier of the serial port, without reference to the module.
serialport→get_hardwareId() Returns the unique hardware identifier of the serial port in the form `SERIAL.FUNCTIONID`.
serialport→get_lastMsg()

Returns the latest message fully received (for Line, Frame and Modbus protocols).

serialport→get_logicalName()

Returns the logical name of the serial port.

serialport→get_module()

Gets the `YModule` object for the device on which the function is located.

serialport→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

serialport→get_msgCount()

Returns the total number of messages received since last reset.

serialport→get_protocol()

Returns the type of protocol used over the serial line, as a string.

serialport→get_rxCount()

Returns the total number of bytes received since last reset.

serialport→get_serialMode()

Returns the serial port communication parameters, as a string such as "9600,8N1".

serialport→get_txCount()

Returns the total number of bytes transmitted since last reset.

serialport→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

serialport→isOnline()

Checks if the serial port is currently reachable, without raising any error.

serialport→isOnline_async(**callback**, **context**)

Checks if the serial port is currently reachable, without raising any error (asynchronous version).

serialport→load(**msValidity**)

Preloads the serial port cache with a specified validity duration.

serialport→load_async(**msValidity**, **callback**, **context**)

Preloads the serial port cache with a specified validity duration (asynchronous version).

serialport→modbusReadBits(**slaveNo**, **pduAddr**, **nBits**)

Reads one or more contiguous internal bits (or coil status) from a MODBUS serial device.

serialport→modbusReadInputBits(**slaveNo**, **pduAddr**, **nBits**)

Reads one or more contiguous input bits (or discrete inputs) from a MODBUS serial device.

serialport→modbusReadInputRegisters(**slaveNo**, **pduAddr**, **nWords**)

Reads one or more contiguous input registers (read-only registers) from a MODBUS serial device.

serialport→modbusReadRegisters(**slaveNo**, **pduAddr**, **nWords**)

Reads one or more contiguous internal registers (holding registers) from a MODBUS serial device.

serialport→modbusWriteAndReadRegisters(**slaveNo**, **pduWriteAddr**, **values**, **pduReadAddr**, **nReadWords**)

Sets several contiguous internal registers (holding registers) on a MODBUS serial device, then performs a contiguous read of a set of (possibly different) internal registers.

serialport→modbusWriteBit(**slaveNo**, **pduAddr**, **value**)

Sets a single internal bit (or coil) on a MODBUS serial device.

serialport→modbusWriteBits(**slaveNo**, **pduAddr**, **bits**)

Sets several contiguous internal bits (or coils) on a MODBUS serial device.

serialport→modbusWriteRegister(**slaveNo**, **pduAddr**, **value**)

Sets a single internal register (or holding register) on a MODBUS serial device.

serialport→modbusWriteRegisters(**slaveNo**, **pduAddr**, **values**)

Sets several contiguous internal registers (or holding registers) on a MODBUS serial device.

serialport→nextSerialPort()

Continues the enumeration of serial ports started using `yFirstSerialPort()`.

serialport→queryLine(query, maxWait)

Sends a text line query to the serial port, and reads the reply, if any.

serialport→queryMODBUS(slaveNo, pduBytes)

Sends a message to a specified MODBUS slave connected to the serial port, and reads the reply, if any.

serialport→readHex(nBytes)

Reads data from the receive buffer as a hexadecimal string, starting at current stream position.

serialport→readLine()

Reads a single line (or message) from the receive buffer, starting at current stream position.

serialport→readMessages(pattern, maxWait)

Searches for incoming messages in the serial port receive buffer matching a given pattern, starting at current position.

serialport→readStr(nChars)

Reads data from the receive buffer as a string, starting at current stream position.

serialport→read_seek(rxCountVal)

Changes the current internal stream position to the specified value.

serialport→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

serialport→reset()

Clears the serial port buffer and resets counters to zero.

serialport→set_RTS(val)

Manually sets the state of the RTS line.

serialport→set_logicalName(newval)

Changes the logical name of the serial port.

serialport→set_protocol(newval)

Changes the type of protocol used over the serial line.

serialport→set_serialMode(newval)

Changes the serial port communication parameters, with a string such as "9600,8N1".

serialport→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

serialport→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

serialport→writeArray(byteList)

Sends a byte sequence (provided as a list of bytes) to the serial port.

serialport→writeBin(buff)

Sends a binary buffer to the serial port, as is.

serialport→writeHex(hexString)

Sends a byte sequence (provided as a hexadecimal string) to the serial port.

serialport→writeLine(text)

Sends an ASCII string to the serial port, followed by a line break (CR LF).

serialport→writeMODBUS(hexString)

Sends a MODBUS message (provided as a hexadecimal string) to the serial port.

serialport→writeStr(text)

Sends an ASCII string to the serial port, as is.

serialport→get_CTS() YSerialPort
serialport→CTS()YSerialPort get_CTS

Read the level of the CTS line.

```
YSerialPort target get_CTS
```

The CTS line is usually driven by the RTS signal of the connected serial device.

Returns :
1 if the CTS line is high, 0 if the CTS line is low.

On failure, throws an exception or returns a negative error code.

serialport→get_advertisedValue() YSerialPort
serialport→advertisedValue()YSerialPort
get_advertisedValue

Returns the current value of the serial port (no more than 6 characters).

YSerialPort **target get_advertisedValue**

Returns :
a string corresponding to the current value of the serial port (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

serialport→get_errCount() YSerialPort
serialport→errCount()YSerialPort get_errCount

Returns the total number of communication errors detected since last reset.

YSerialPort **target** **get_errCount**

Returns :
an integer corresponding to the total number of communication errors detected since last reset

On failure, throws an exception or returns `Y_ERRCOUNT_INVALID`.

serialport→get_lastMsg() YSerialPort
serialport→lastMsg()YSerialPort get_lastMsg

Returns the latest message fully received (for Line, Frame and Modbus protocols).

YSerialPort **target get_lastMsg**

Returns :
a string corresponding to the latest message fully received (for Line, Frame and Modbus protocols)

On failure, throws an exception or returns `Y_LASTMSG_INVALID`.

serialport→get_logicalName() YSerialPort
serialport→logicalName()YSerialPort
get_logicalName

Returns the logical name of the serial port.

YSerialPort **target get_logicalName**

Returns :
a string corresponding to the logical name of the serial port.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

serialport→get_msgCount() YSerialPort
serialport→msgCount()YSerialPort get_msgCount

Returns the total number of messages received since last reset.

YSerialPort **target get_msgCount**

Returns :
an integer corresponding to the total number of messages received since last reset

On failure, throws an exception or returns `Y_MSGCOUNT_INVALID`.

serialport→get_protocol() YSerialPort
serialport→protocol()YSerialPort get_protocol

Returns the type of protocol used over the serial line, as a string.

```
YSerialPort target get_protocol
```

Possible values are "Line" for ASCII messages separated by CR and/or LF, "Frame:[timeout]ms" for binary messages separated by a delay time, "Modbus-ASCII" for MODBUS messages in ASCII mode, "Modbus-RTU" for MODBUS messages in RTU mode, "Char" for a continuous ASCII stream or "Byte" for a continuous binary stream.

Returns :

a string corresponding to the type of protocol used over the serial line, as a string

On failure, throws an exception or returns `Y_PROTOCOL_INVALID`.

serialport→get_rxCount() YSerialPort
serialport→rxCount()YSerialPort get_rxCount

Returns the total number of bytes received since last reset.

YSerialPort **target get_rxCount**

Returns :
an integer corresponding to the total number of bytes received since last reset

On failure, throws an exception or returns `Y_RXCOUNT_INVALID`.

serialport→get_serialMode() YSerialPort
serialport→serialMode()YSerialPort get_serialMode

Returns the serial port communication parameters, as a string such as "9600,8N1".

YSerialPort **target** **get_serialMode**

The string includes the baud rate, the number of data bits, the parity, and the number of stop bits. An optional suffix is included if flow control is active: "CtsRts" for hardware handshake, "XOnXOff" for logical flow control and "Simplex" for acquiring a shared bus using the RTS line (as used by some RS485 adapters for instance).

Returns :

a string corresponding to the serial port communication parameters, as a string such as "9600,8N1"

On failure, throws an exception or returns `Y_SERIALMODE_INVALID`.

serialport→get_txCount() YSerialPort
serialport→txCount()YSerialPort get_txCount

Returns the total number of bytes transmitted since last reset.

YSerialPort **target get_txCount**

Returns :
an integer corresponding to the total number of bytes transmitted since last reset

On failure, throws an exception or returns `Y_TXCOUNT_INVALID`.

serialport→modbusReadBits()YSerialPort YSerialPort modbusReadBits

Reads one or more contiguous internal bits (or coil status) from a MODBUS serial device.

YSerialPort **target modbusReadBits slaveNo pduAddr nBits**

This method uses the MODBUS function code 0x01 (Read Coils).

Parameters :

slaveNo the address of the slave MODBUS device to query

pduAddr the relative address of the first bit/coil to read (zero-based)

nBits the number of bits/coils to read

Returns :

a vector of integers, each corresponding to one bit.

On failure, throws an exception or returns an empty array.

serialport→modbusReadInputBits()YSerialPort modbusReadInputBits YSerialPort

Reads one or more contiguous input bits (or discrete inputs) from a MODBUS serial device.

YSerialPort **target modbusReadInputBits slaveNo pduAddr nBits**

This method uses the MODBUS function code 0x02 (Read Discrete Inputs).

Parameters :

slaveNo the address of the slave MODBUS device to query

pduAddr the relative address of the first bit/input to read (zero-based)

nBits the number of bits/inputs to read

Returns :

a vector of integers, each corresponding to one bit.

On failure, throws an exception or returns an empty array.

serialport→modbusReadInputRegisters()YSerialPort YSerialPort modbusReadInputRegisters

Reads one or more contiguous input registers (read-only registers) from a MODBUS serial device.

YSerialPort **target modbusReadInputRegisters slaveNo pduAddr nWords**

This method uses the MODBUS function code 0x04 (Read Input Registers).

Parameters :

slaveNo the address of the slave MODBUS device to query

pduAddr the relative address of the first input register to read (zero-based)

nWords the number of input registers to read

Returns :

a vector of integers, each corresponding to one 16-bit input value.

On failure, throws an exception or returns an empty array.

serialport→modbusReadRegisters()YSerialPort modbusReadRegisters YSerialPort

Reads one or more contiguous internal registers (holding registers) from a MODBUS serial device.

YSerialPort **target modbusReadRegisters slaveNo pduAddr nWords**

This method uses the MODBUS function code 0x03 (Read Holding Registers).

Parameters :

slaveNo the address of the slave MODBUS device to query

pduAddr the relative address of the first holding register to read (zero-based)

nWords the number of holding registers to read

Returns :

a vector of integers, each corresponding to one 16-bit register value.

On failure, throws an exception or returns an empty array.

serialport→modbusWriteAndReadRegisters() YSerialPort
YSerialPort modbusWriteAndReadRegisters

Sets several contiguous internal registers (holding registers) on a MODBUS serial device, then performs a contiguous read of a set of (possibly different) internal registers.

YSerialPort **target modbusWriteAndReadRegisters slaveNo pduWriteAddr values pduReadAddr nReadWords**

This method uses the MODBUS function code 0x17 (Read/Write Multiple Registers).

Parameters :

slaveNo	the address of the slave MODBUS device to drive
pduWriteAddr	the relative address of the first internal register to set (zero-based)
values	the vector of 16 bit values to set
pduReadAddr	the relative address of the first internal register to read (zero-based)
nReadWords	the number of 16 bit values to read

Returns :

a vector of integers, each corresponding to one 16-bit register value read.

On failure, throws an exception or returns an empty array.

serialport→modbusWriteBit()YSerialPort modbusWriteBit YSerialPort

Sets a single internal bit (or coil) on a MODBUS serial device.

YSerialPort **target modbusWriteBit slaveNo pduAddr value**

This method uses the MODBUS function code 0x05 (Write Single Coil).

Parameters :

slaveNo the address of the slave MODBUS device to drive

pduAddr the relative address of the bit/coil to set (zero-based)

value the value to set (0 for OFF state, non-zero for ON state)

Returns :

the number of bits/coils affected on the device (1)

On failure, throws an exception or returns zero.

serialport→modbusWriteBits()YSerialPort YSerialPort modbusWriteBits

Sets several contiguous internal bits (or coils) on a MODBUS serial device.

```
YSerialPort target modbusWriteBits slaveNo pduAddr bits
```

This method uses the MODBUS function code 0x0f (Write Multiple Coils).

Parameters :

slaveNo	the address of the slave MODBUS device to drive
pduAddr	the relative address of the first bit/coil to set (zero-based)
bits	the vector of bits to be set (one integer per bit)

Returns :

the number of bits/coils affected on the device

On failure, throws an exception or returns zero.

serialport→modbusWriteRegister()YSerialPort modbusWriteRegister YSerialPort

Sets a single internal register (or holding register) on a MODBUS serial device.

YSerialPort **target modbusWriteRegister slaveNo pduAddr value**

This method uses the MODBUS function code 0x06 (Write Single Register).

Parameters :

slaveNo the address of the slave MODBUS device to drive

pduAddr the relative address of the register to set (zero-based)

value the 16 bit value to set

Returns :

the number of registers affected on the device (1)

On failure, throws an exception or returns zero.

serialport→modbusWriteRegisters()YSerialPort YSerialPort modbusWriteRegisters

Sets several contiguous internal registers (or holding registers) on a MODBUS serial device.

```
YSerialPort target modbusWriteRegisters slaveNo pduAddr values
```

This method uses the MODBUS function code 0x10 (Write Multiple Registers).

Parameters :

slaveNo the address of the slave MODBUS device to drive

pduAddr the relative address of the first internal register to set (zero-based)

values the vector of 16 bit values to set

Returns :

the number of registers affected on the device

On failure, throws an exception or returns zero.

serialport→queryLine()YSerialPort queryLine YSerialPort

Sends a text line query to the serial port, and reads the reply, if any.

```
YSerialPort target queryLine query maxWait
```

This function can only be used when the serial port is configured for 'Line' protocol.

Parameters :

query the line query to send (without CR/LF)

maxWait the maximum number of milliseconds to wait for a reply.

Returns :

the next text line received after sending the text query, as a string. Additional lines can be obtained by calling readLine or readMessages.

On failure, throws an exception or returns an empty array.

serialport→queryMODBUS()YSerialPort YSerialPort queryMODBUS

Sends a message to a specified MODBUS slave connected to the serial port, and reads the reply, if any.

```
YSerialPort target queryMODBUS slaveNo pduBytes
```

The message is the PDU, provided as a vector of bytes.

Parameters :

slaveNo the address of the slave MODBUS device to query

pduBytes the message to send (PDU), as a vector of bytes. The first byte of the PDU is the MODBUS function code.

Returns :

the received reply, as a vector of bytes.

On failure, throws an exception or returns an empty array (or a MODBUS error reply).

serialport→readHex()YSerialPort readHex YSerialPort

Reads data from the receive buffer as a hexadecimal string, starting at current stream position.

```
YSerialPort target readHex nBytes
```

If data at current stream position is not available anymore in the receive buffer, the function performs a short read.

Parameters :

nBytes the maximum number of bytes to read

Returns :

a string with receive buffer contents, encoded in hexadecimal

On failure, throws an exception or returns a negative error code.

serialport→readLine()YSerialPort readLine YSerialPort

Reads a single line (or message) from the receive buffer, starting at current stream position.

YSerialPort **target readLine**

This function can only be used when the serial port is configured for a message protocol, such as 'Line' mode or MODBUS protocols. It does not work in plain stream modes, eg. 'Char' or 'Byte').

If data at current stream position is not available anymore in the receive buffer, the function returns the oldest available line and moves the stream position just after. If no new full line is received, the function returns an empty line.

Returns :
a string with a single line of text

On failure, throws an exception or returns a negative error code.

serialport→readMessages()YSerialPort YSerialPort readMessages

Searches for incoming messages in the serial port receive buffer matching a given pattern, starting at current position.

YSerialPort **target readMessages pattern maxWait**

This function can only be used when the serial port is configured for a message protocol, such as 'Line' mode or MODBUS protocols. It does not work in plain stream modes, eg. 'Char' or 'Byte', for which there is no "start" of message.

The search returns all messages matching the expression provided as argument in the buffer. If no matching message is found, the search waits for one up to the specified maximum timeout (in milliseconds).

Parameters :

pattern a limited regular expression describing the expected message format, or an empty string if all messages should be returned (no filtering). When using binary protocols, the format applies to the hexadecimal representation of the message.

maxWait the maximum number of milliseconds to wait for a message if none is found in the receive buffer.

Returns :

an array of strings containing the messages found, if any. Binary messages are converted to hexadecimal representation.

On failure, throws an exception or returns an empty array.

serialport→readStr()YSerialPort readStr YSerialPort

Reads data from the receive buffer as a string, starting at current stream position.

```
YSerialPort target readStr nChars
```

If data at current stream position is not available anymore in the receive buffer, the function performs a short read.

Parameters :

nChars the maximum number of characters to read

Returns :

a string with receive buffer contents

On failure, throws an exception or returns a negative error code.

serialport→read_seek()YSerialPort read_seek YSerialPort

Changes the current internal stream position to the specified value.

```
YSerialPort target read_seek rxCountVal
```

This function does not affect the device, it only changes the value stored in the YSerialPort object for the next read operations.

Parameters :

rxCountVal the absolute position index (value of rxCount) for next read operations.

Returns :

nothing.

serialport→reset()YSerialPort reset YSerialPort

Clears the serial port buffer and resets counters to zero.

YSerialPort **target reset**

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→set_RTS() YSerialPort
serialport→setRTS()YSerialPort set_RTS

Manually sets the state of the RTS line.

```
YSerialPort target set_RTS val
```

This function has no effect when hardware handshake is enabled, as the RTS line is driven automatically.

Parameters :

val 1 to turn RTS on, 0 to turn RTS off

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→set_logicalName() YSerialPort
serialport→setLogicalName()YSerialPort
set_logicalName

Changes the logical name of the serial port.

```
YSerialPort 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 serial port.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→set_protocol() YSerialPort
serialport→setProtocol()YSerialPort set_protocol

Changes the type of protocol used over the serial line.

```
YSerialPort target set_protocol newval
```

Possible values are "Line" for ASCII messages separated by CR and/or LF, "Frame:[timeout]ms" for binary messages separated by a delay time, "Modbus-ASCII" for MODBUS messages in ASCII mode, "Modbus-RTU" for MODBUS messages in RTU mode, "Char" for a continuous ASCII stream or "Byte" for a continuous binary stream.

Parameters :

newval a string corresponding to the type of protocol used over the serial line

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→set_serialMode() YSerialPort
serialport→setSerialMode()YSerialPort
set_serialMode

Changes the serial port communication parameters, with a string such as "9600,8N1".

```
YSerialPort target set_serialMode newval
```

The string includes the baud rate, the number of data bits, the parity, and the number of stop bits. An optional suffix can be added to enable flow control: "CtsRts" for hardware handshake, "XOnXOff" for logical flow control and "Simplex" for acquiring a shared bus using the RTS line (as used by some RS485 adapters for instance).

Parameters :

newval a string corresponding to the serial port communication parameters, with a string such as "9600,8N1"

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→writeArray()YSerialPort writeArray YSerialPort

Sends a byte sequence (provided as a list of bytes) to the serial port.

```
YSerialPort target writeArray byteList
```

Parameters :

byteList a list of byte codes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→writeBin()YSerialPort writeBin YSerialPort

Sends a binary buffer to the serial port, as is.

```
YSerialPort target writeBin buff
```

Parameters :

buff the binary buffer to send

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→writeHex()YSerialPort writeHex YSerialPort

Sends a byte sequence (provided as a hexadecimal string) to the serial port.

```
YSerialPort target writeHex hexString
```

Parameters :

hexString a string of hexadecimal byte codes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→writeLine()YSerialPort writeLine YSerialPort

Sends an ASCII string to the serial port, followed by a line break (CR LF).

```
YSerialPort target writeLine text
```

Parameters :

text the text string to send

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→writeMODBUS()YSerialPort writeMODBUS

YSerialPort

Sends a MODBUS message (provided as a hexadecimal string) to the serial port.

YSerialPort **target writeMODBUS hexString**

The message must start with the slave address. The MODBUS CRC/LRC is automatically added by the function. This function does not wait for a reply.

Parameters :

hexString a hexadecimal message string, including device address but no CRC/LRC

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→writeStr()YSerialPort writeStr YSerialPort

Sends an ASCII string to the serial port, as is.

```
YSerialPort target writeStr text
```

Parameters :

text the text string to send

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.41. Servo function interface

Yoctopuce application programming interface allows you not only to move a servo to a given position, but also to specify the time interval in which the move should be performed. This makes it possible to synchronize two servos involved in a same move.

In order to use the functions described here, you should include:

```
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');
cpp     #include "yocto_servo.h"
m       #import "yocto_servo.h"
pas     uses yocto_servo;
vb      yocto_servo.vb
cs      yocto_servo.cs
java    import com.yoctopuce.YoctoAPI.YServo;
py      from yocto_servo import *
```

Global functions
yFindServo(**func**) Retrieves a servo for a given identifier.
yFirstServo() Starts the enumeration of servos currently accessible.
YServo methods
servo→describe() Returns a short text that describes unambiguously the instance of the servo in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
servo→get_advertisedValue() Returns the current value of the servo (no more than 6 characters).
servo→get_enabled() Returns the state of the servos.
servo→get_enabledAtPowerOn() Returns the servo signal generator state at power up.
servo→get_errorMessage() Returns the error message of the latest error with the servo.
servo→get_errorType() Returns the numerical error code of the latest error with the servo.
servo→get_friendlyName() Returns a global identifier of the servo in the format `MODULE_NAME.FUNCTION_NAME`.
servo→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
servo→get_functionId() Returns the hardware identifier of the servo, without reference to the module.
servo→get_hardwareId() Returns the unique hardware identifier of the servo in the form `SERIAL.FUNCTIONID`.
servo→get_logicalName() Returns the logical name of the servo.

servo→get_module()

Gets the `YModule` object for the device on which the function is located.

servo→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

servo→get_neutral()

Returns the duration in microseconds of a neutral pulse for the servo.

servo→get_position()

Returns the current servo position.

servo→get_positionAtPowerOn()

Returns the servo position at device power up.

servo→get_range()

Returns the current range of use of the servo.

servo→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

servo→isOnline()

Checks if the servo is currently reachable, without raising any error.

servo→isOnline_async(**callback**, **context**)

Checks if the servo is currently reachable, without raising any error (asynchronous version).

servo→load(**msValidity**)

Preloads the servo cache with a specified validity duration.

servo→load_async(**msValidity**, **callback**, **context**)

Preloads the servo cache with a specified validity duration (asynchronous version).

servo→move(**target**, **ms_duration**)

Performs a smooth move at constant speed toward a given position.

servo→nextServo()

Continues the enumeration of servos started using `yFirstServo()`.

servo→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

servo→set_enabled(**newval**)

Stops or starts the servo.

servo→set_enabledAtPowerOn(**newval**)

Configure the servo signal generator state at power up.

servo→set_logicalName(**newval**)

Changes the logical name of the servo.

servo→set_neutral(**newval**)

Changes the duration of the pulse corresponding to the neutral position of the servo.

servo→set_position(**newval**)

Changes immediately the servo driving position.

servo→set_positionAtPowerOn(**newval**)

Configure the servo position at device power up.

servo→set_range(**newval**)

Changes the range of use of the servo, specified in per cents.

servo→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

servo→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

servo→get_advertisedValue() YServo
servo→advertisedValue()YServo
get_advertisedValue

Returns the current value of the servo (no more than 6 characters).

YServo **target** **get_advertisedValue**

Returns :
a string corresponding to the current value of the servo (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

servo→get_enabled() YServo
servo→enabled()YServo get_enabled

Returns the state of the servos.

```
YServo target get_enabled
```

Returns :

either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to the state of the servos

On failure, throws an exception or returns Y_ENABLED_INVALID.

servo→get_enabledAtPowerOn() YServo
servo→enabledAtPowerOn()YServo
get_enabledAtPowerOn

Returns the servo signal generator state at power up.

YServo **target** **get_enabledAtPowerOn**

Returns :

either `Y_ENABLEDATPOWERON_FALSE` or `Y_ENABLEDATPOWERON_TRUE`, according to the servo signal generator state at power up

On failure, throws an exception or returns `Y_ENABLEDATPOWERON_INVALID`.

servo→get_logicalName() YServo
servo→logicalName()YServo get_logicalName

Returns the logical name of the servo.

```
YServo target get_logicalName
```

Returns :
a string corresponding to the logical name of the servo.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

servo→get_neutral() YServo
servo→neutral()YServo get_neutral

Returns the duration in microseconds of a neutral pulse for the servo.

YServo **target get_neutral**

Returns :

an integer corresponding to the duration in microseconds of a neutral pulse for the servo

On failure, throws an exception or returns `Y_NEUTRAL_INVALID`.

servo→get_position() YServo servo→position()YServo get_position

Returns the current servo position.

```
YServo target get_position
```

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()YServo get_positionAtPowerOn

Returns the servo position at device power up.

YServo **target get_positionAtPowerOn**

Returns :
an integer corresponding to the servo position at device power up

On failure, throws an exception or returns `Y_POSITIONATPOWERON_INVALID`.

servo→get_range() YServo
servo→range()YServo get_range

Returns the current range of use of the servo.

```
YServo target get_range
```

Returns :
an integer corresponding to the current range of use of the servo

On failure, throws an exception or returns `Y_RANGE_INVALID`.

servo→move()YServo move YServo

Performs a smooth move at constant speed toward a given position.

YServo **target move target ms_duration**

Parameters :

target	new position at the end of the move
ms_duration	total duration of the move, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_enabled() YServo
servo→setEnabled()YServo set_enabled

Stops or starts the servo.

YServo **target** **set_enabled** **newval**

Parameters :

newval either Y_ENABLED_FALSE or Y_ENABLED_TRUE

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_enabledAtPowerOn() YServo
servo→setEnabledAtPowerOn()YServo
set_enabledAtPowerOn

Configure the servo signal generator state at power up.

YServo **target set_enabledAtPowerOn newval**

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval either `Y_ENABLEDATPOWERON_FALSE` or `Y_ENABLEDATPOWERON_TRUE`

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_logicalName() YServo
servo→setLogicalName()YServo set_logicalName

Changes the logical name of the servo.

```
YServo 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 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()YServo set_neutral

Changes the duration of the pulse corresponding to the neutral position of the servo.

```
YServo target set_neutral newval
```

The duration is specified in microseconds, and the standard value is 1500 [us]. This setting makes it possible to shift the range of use of the servo. Be aware that using a range higher than what is supported by the servo is likely to damage the servo.

Parameters :

newval an integer corresponding to the duration of the pulse corresponding to the neutral position of the servo

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_position() YServo
servo→setPosition()YServo set_position

Changes immediately the servo driving position.

```
YServo target set_position newval
```

Parameters :

newval an integer corresponding to immediately the servo driving position

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_positionAtPowerOn() YServo servo→setPositionAtPowerOn()YServo set_positionAtPowerOn

Configure the servo position at device power up.

```
YServo target set_positionAtPowerOn newval
```

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval an integer

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

servo→set_range() YServo
servo→setRange()YServo set_range

Changes the range of use of the servo, specified in per cents.

```
YServo target set_range newval
```

A range of 100% corresponds to a standard control signal, that varies from 1 [ms] to 2 [ms], When using a servo that supports a double range, from 0.5 [ms] to 2.5 [ms], you can select a range of 200%. Be aware that using a range higher than what is supported by the servo is likely to damage the servo.

Parameters :

newval an integer corresponding to the range of use of the servo, specified in per cents

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.42. Temperature function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_temperature.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YTemperature = yoctolib.YTemperature;
php     require_once('yocto_temperature.php');
cpp     #include "yocto_temperature.h"
m       #import "yocto_temperature.h"
pas     uses yocto_temperature;
vb      yocto_temperature.vb
cs      yocto_temperature.cs
java    import com.yoctopuce.YoctoAPI.YTemperature;
py      from yocto_temperature import *
```

Global functions
yFindTemperature(**func**) Retrieves a temperature sensor for a given identifier.
yFirstTemperature() Starts the enumeration of temperature sensors currently accessible.
`YTemperature` methods
temperature→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
temperature→describe() Returns a short text that describes unambiguously the instance of the temperature sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
temperature→get_advertisedValue() Returns the current value of the temperature sensor (no more than 6 characters).
temperature→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in Celsius, as a floating point number.
temperature→get_currentValue() Returns the current value of the temperature, in Celsius, as a floating point number.
temperature→get_errorMessage() Returns the error message of the latest error with the temperature sensor.
temperature→get_errorType() Returns the numerical error code of the latest error with the temperature sensor.
temperature→get_friendlyName() Returns a global identifier of the temperature sensor in the format `MODULE_NAME.FUNCTION_NAME`.
temperature→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
temperature→get_functionId() Returns the hardware identifier of the temperature sensor, without reference to the module.
temperature→get_hardwareId() Returns the unique hardware identifier of the temperature sensor in the form `SERIAL.FUNCTIONID`.

temperature→get_highestValue()

Returns the maximal value observed for the temperature since the device was started.

temperature→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

temperature→get_logicalName()

Returns the logical name of the temperature sensor.

temperature→get_lowestValue()

Returns the minimal value observed for the temperature since the device was started.

temperature→get_module()

Gets the `YModule` object for the device on which the function is located.

temperature→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

temperature→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

temperature→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

temperature→get_resolution()

Returns the resolution of the measured values.

temperature→get_sensorType()

Returns the temperature sensor type.

temperature→get_unit()

Returns the measuring unit for the temperature.

temperature→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

temperature→isOnline()

Checks if the temperature sensor is currently reachable, without raising any error.

temperature→isOnline_async(**callback**, **context**)

Checks if the temperature sensor is currently reachable, without raising any error (asynchronous version).

temperature→load(**msValidity**)

Preloads the temperature sensor cache with a specified validity duration.

temperature→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

temperature→load_async(**msValidity**, **callback**, **context**)

Preloads the temperature sensor cache with a specified validity duration (asynchronous version).

temperature→nextTemperature()

Continues the enumeration of temperature sensors started using `yFirstTemperature()`.

temperature→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

temperature→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

temperature→set_highestValue(**newval**)

Changes the recorded maximal value observed.

temperature→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

temperature→set_logicalName(newval)

Changes the logical name of the temperature sensor.

temperature→set_lowestValue(newval)

Changes the recorded minimal value observed.

temperature→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

temperature→set_resolution(newval)

Changes the resolution of the measured physical values.

temperature→set_sensorType(newval)

Modify the temperature sensor type.

temperature→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

temperature→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

temperature→calibrateFromPoints()YTemperature YTemperature calibrateFromPoints

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

YTemperature **target calibrateFromPoints rawValues refValues**

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→get_advertisedValue() YTemperature
temperature→advertisedValue()YTemperature
get_advertisedValue

Returns the current value of the temperature sensor (no more than 6 characters).

YTemperature **target get_advertisedValue**

Returns :
a string corresponding to the current value of the temperature sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

temperature→get_currentRawValue() YTemperature
temperature→currentRawValue()YTemperature
get_currentRawValue

Returns the uncalibrated, unrounded raw value returned by the sensor, in Celsius, as a floating point number.

YTemperature **target** **get_currentRawValue**

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in Celsius, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

temperature→get_currentValue() YTemperature
temperature→currentValue()YTemperature
get_currentValue

Returns the current value of the temperature, in Celsius, as a floating point number.

```
YTemperature target get_currentValue
```

Returns :

a floating point number corresponding to the current value of the temperature, in Celsius, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

temperature→get_highestValue() YTemperature
temperature→highestValue()YTemperature
get_highestValue

Returns the maximal value observed for the temperature since the device was started.

YTemperature **target get_highestValue**

Returns :
a floating point number corresponding to the maximal value observed for the temperature since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

temperature→get_logFrequency() YTemperature
temperature→logFrequency()YTemperature
get_logFrequency

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

YTemperature **target get_logFrequency**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

temperature→get_logicalName() YTemperature
temperature→logicalName()YTemperature
get_logicalName

Returns the logical name of the temperature sensor.

```
YTemperature target get_logicalName
```

Returns :
a string corresponding to the logical name of the temperature sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

temperature→get_lowestValue() YTemperature
temperature→lowestValue()YTemperature
get_lowestValue

Returns the minimal value observed for the temperature since the device was started.

YTemperature **target get_lowestValue**

Returns :

a floating point number corresponding to the minimal value observed for the temperature since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

temperature→get_recordedData() YTemperature
temperature→recordedData()YTemperature
get_recordedData

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
YTemperature target get_recordedData startTime endTime
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

temperature→get_reportFrequency() YTemperature
temperature→reportFrequency()YTemperature
get_reportFrequency

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

YTemperature **target get_reportFrequency**

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

temperature→get_resolution() YTemperature
temperature→resolution()YTemperature
get_resolution

Returns the resolution of the measured values.

```
YTemperature target get_resolution
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

temperature→get_sensorType() YTemperature
temperature→sensorType()YTemperature
get_sensorType

Returns the temperature sensor type.

YTemperature **target get_sensorType**

Returns :

a value among Y_SENSORTYPE_DIGITAL, Y_SENSORTYPE_TYPE_K, Y_SENSORTYPE_TYPE_E, Y_SENSORTYPE_TYPE_J, Y_SENSORTYPE_TYPE_N, Y_SENSORTYPE_TYPE_R, Y_SENSORTYPE_TYPE_S, Y_SENSORTYPE_TYPE_T, Y_SENSORTYPE_PT100_4WIRES, Y_SENSORTYPE_PT100_3WIRES and Y_SENSORTYPE_PT100_2WIRES corresponding to the temperature sensor type

On failure, throws an exception or returns Y_SENSORTYPE_INVALID.

temperature→get_unit() YTemperature
temperature→unit()YTemperature get_unit

Returns the measuring unit for the temperature.

```
YTemperature target get_unit
```

Returns :
a string corresponding to the measuring unit for the temperature

On failure, throws an exception or returns `Y_UNIT_INVALID`.

temperature→loadCalibrationPoints()YTemperature YTemperature loadCalibrationPoints

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

YTemperature **target loadCalibrationPoints rawValues refValues**

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_highestValue() YTemperature
temperature→setHighestValue()YTemperature
set_highestValue

Changes the recorded maximal value observed.

```
YTemperature target set_highestValue newval
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_logFrequency() YTemperature
temperature→setLogFrequency()YTemperature
set_logFrequency

Changes the datalogger recording frequency for this function.

```
YTemperature target set_logFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_logicalName() YTemperature
temperature→setLogicalName()YTemperature
set_logicalName

Changes the logical name of the temperature sensor.

```
YTemperature 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 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()YTemperature
set_lowestValue

Changes the recorded minimal value observed.

```
YTemperature target set_lowestValue newval
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_reportFrequency() YTemperature
temperature→setReportFrequency()YTemperature
set_reportFrequency

Changes the timed value notification frequency for this function.

YTemperature **target set_reportFrequency newval**

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_resolution() YTemperature
temperature→setResolution()YTemperature
set_resolution

Changes the resolution of the measured physical values.

```
YTemperature target set_resolution newval
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

temperature→set_sensorType() YTemperature
temperature→setSensorType()YTemperature
set_sensorType

Modify the temperature sensor type.

YTemperature **target set_sensorType newval**

This function is used to to define the type of thermocouple (K,E...) used with the device. This will have no effect if module is using a digital sensor. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a value among `Y_SENSORTYPE_DIGITAL`, `Y_SENSORTYPE_TYPE_K`, `Y_SENSORTYPE_TYPE_E`, `Y_SENSORTYPE_TYPE_J`, `Y_SENSORTYPE_TYPE_N`, `Y_SENSORTYPE_TYPE_R`, `Y_SENSORTYPE_TYPE_S`, `Y_SENSORTYPE_TYPE_T`, `Y_SENSORTYPE_PT100_4WIRES`, `Y_SENSORTYPE_PT100_3WIRES` and `Y_SENSORTYPE_PT100_2WIRES`

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.43. Tilt function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_tilt.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YTilt = yoctolib.YTilt;
php     require_once('yocto_tilt.php');
cpp     #include "yocto_tilt.h"
m       #import "yocto_tilt.h"
pas     uses yocto_tilt;
vb      yocto_tilt.vb
cs      yocto_tilt.cs
java    import com.yoctopuce.YoctoAPI.YTilt;
py      from yocto_tilt import *
```

Global functions
yFindTilt(**func**) Retrieves a tilt sensor for a given identifier.
yFirstTilt() Starts the enumeration of tilt sensors currently accessible.
YTilt methods
tilt→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
tilt→describe() Returns a short text that describes unambiguously the instance of the tilt sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
tilt→get_advertisedValue() Returns the current value of the tilt sensor (no more than 6 characters).
tilt→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in degrees, as a floating point number.
tilt→get_currentValue() Returns the current value of the inclination, in degrees, as a floating point number.
tilt→get_errorMessage() Returns the error message of the latest error with the tilt sensor.
tilt→get_errorType() Returns the numerical error code of the latest error with the tilt sensor.
tilt→get_friendlyName() Returns a global identifier of the tilt sensor in the format `MODULE_NAME.FUNCTION_NAME`.
tilt→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
tilt→get_functionId() Returns the hardware identifier of the tilt sensor, without reference to the module.
tilt→get_hardwareId() Returns the unique hardware identifier of the tilt sensor in the form `SERIAL.FUNCTIONID`.

tilt→get_highestValue()

Returns the maximal value observed for the inclination since the device was started.

tilt→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

tilt→get_logicalName()

Returns the logical name of the tilt sensor.

tilt→get_lowestValue()

Returns the minimal value observed for the inclination since the device was started.

tilt→get_module()

Gets the `YModule` object for the device on which the function is located.

tilt→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

tilt→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

tilt→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

tilt→get_resolution()

Returns the resolution of the measured values.

tilt→get_unit()

Returns the measuring unit for the inclination.

tilt→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

tilt→isOnline()

Checks if the tilt sensor is currently reachable, without raising any error.

tilt→isOnline_async(**callback**, **context**)

Checks if the tilt sensor is currently reachable, without raising any error (asynchronous version).

tilt→load(**msValidity**)

Preloads the tilt sensor cache with a specified validity duration.

tilt→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

tilt→load_async(**msValidity**, **callback**, **context**)

Preloads the tilt sensor cache with a specified validity duration (asynchronous version).

tilt→nextTilt()

Continues the enumeration of tilt sensors started using `yFirstTilt()`.

tilt→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

tilt→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

tilt→set_highestValue(**newval**)

Changes the recorded maximal value observed.

tilt→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

tilt→set_logicalName(**newval**)

Changes the logical name of the tilt sensor.

tilt→set_lowestValue(**newval**)

Changes the recorded minimal value observed.

tilt→set_reportFrequency(**newval**)

Changes the timed value notification frequency for this function.

tilt→set_resolution(**newval**)

Changes the resolution of the measured physical values.

tilt→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

tilt→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

tilt→calibrateFromPoints()YTilt calibrateFromPoints YTilt

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

YTilt **target calibrateFromPoints rawValues refValues**

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→get_advertisedValue() YTilt
tilt→advertisedValue()YTilt get_advertisedValue

Returns the current value of the tilt sensor (no more than 6 characters).

YTilt **target get_advertisedValue**

Returns :

a string corresponding to the current value of the tilt sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

tilt→get_currentRawValue() YTilt
tilt→currentRawValue()YTilt get_currentRawValue

Returns the uncalibrated, unrounded raw value returned by the sensor, in degrees, as a floating point number.

```
YTilt target get_currentRawValue
```

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in degrees, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

tilt→get_currentValue() YTilt
tilt→currentValue()YTilt get_currentValue

Returns the current value of the inclination, in degrees, as a floating point number.

YTilt **target get_currentValue**

Returns :

a floating point number corresponding to the current value of the inclination, in degrees, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

tilt→get_highestValue() YTilt
tilt→highestValue()YTilt get_highestValue

Returns the maximal value observed for the inclination since the device was started.

YTilt **target** **get_highestValue**

Returns :

a floating point number corresponding to the maximal value observed for the inclination since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

tilt→get_logFrequency() YTilt
tilt→logFrequency()YTilt get_logFrequency

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

YTilt **target get_logFrequency**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

tilt→get_logicalName() YTilt
tilt→logicalName()YTilt get_logicalName

Returns the logical name of the tilt sensor.

```
YTilt target get_logicalName
```

Returns :
a string corresponding to the logical name of the tilt sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

tilt→get_lowestValue() YTilt
tilt→lowestValue()YTilt get_lowestValue

Returns the minimal value observed for the inclination since the device was started.

YTilt **target get_lowestValue**

Returns :

a floating point number corresponding to the minimal value observed for the inclination since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

tilt→get_recordedData() YTilt
tilt→recordedData()YTilt get_recordedData

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YTilt **target get_recordedData startTime endTime**

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

tilt→get_reportFrequency() YTilt
tilt→reportFrequency()YTilt get_reportFrequency

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

```
YTilt target get_reportFrequency
```

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

tilt→get_resolution() YTilt
tilt→resolution()YTilt get_resolution

Returns the resolution of the measured values.

```
YTilt target get_resolution
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

tilt→get_unit() YTilt
tilt→unit()YTilt get_unit

Returns the measuring unit for the inclination.

```
YTilt target get_unit
```

Returns :

a string corresponding to the measuring unit for the inclination

On failure, throws an exception or returns `Y_UNIT_INVALID`.

tilt→loadCalibrationPoints()YTilt loadCalibrationPoints YTilt

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

YTilt **target loadCalibrationPoints rawValues refValues**

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_highestValue() YTilt
tilt→setHighestValue()YTilt set_highestValue

Changes the recorded maximal value observed.

```
YTilt target set_highestValue newval
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_logFrequency() YTilt
tilt→setLogFrequency()YTilt set_logFrequency

Changes the datalogger recording frequency for this function.

```
YTilt target set_logFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_logicalName() YTilt
tilt→setLogicalName()YTilt set_logicalName

Changes the logical name of the tilt sensor.

```
YTilt 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 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()YTilt set_lowestValue

Changes the recorded minimal value observed.

```
YTilt target set_lowestValue newval
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_reportFrequency() YTilt
tilt→setReportFrequency()YTilt set_reportFrequency

Changes the timed value notification frequency for this function.

```
YTilt target set_reportFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

tilt→set_resolution() YTilt
tilt→setResolution()YTilt set_resolution

Changes the resolution of the measured physical values.

```
YTilt target set_resolution newval
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.44. Voc function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_voc.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YVoc = yoctolib.YVoc;
php     require_once('yocto_voc.php');
cpp     #include "yocto_voc.h"
m       #import "yocto_voc.h"
pas     uses yocto_voc;
vb      yocto_voc.vb
cs      yocto_voc.cs
java    import com.yoctopuce.YoctoAPI.YVoc;
py      from yocto_voc import *
```

Global functions

yFindVoc(**func**)

Retrieves a Volatile Organic Compound sensor for a given identifier.

yFirstVoc()

Starts the enumeration of Volatile Organic Compound sensors currently accessible.

YVoc methods

voc→calibrateFromPoints(**rawValues**, **refValues**)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

voc→describe()

Returns a short text that describes unambiguously the instance of the Volatile Organic Compound sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

voc→get_advertisedValue()

Returns the current value of the Volatile Organic Compound sensor (no more than 6 characters).

voc→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in ppm (vol), as a floating point number.

voc→get_currentValue()

Returns the current value of the estimated VOC concentration, in ppm (vol), as a floating point number.

voc→get_errorMessage()

Returns the error message of the latest error with the Volatile Organic Compound sensor.

voc→get_errorType()

Returns the numerical error code of the latest error with the Volatile Organic Compound sensor.

voc→get_friendlyName()

Returns a global identifier of the Volatile Organic Compound sensor in the format `MODULE_NAME.FUNCTION_NAME`.

voc→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

voc→get_functionId()

Returns the hardware identifier of the Volatile Organic Compound sensor, without reference to the module.

voc→get_hardwareId()

Returns the unique hardware identifier of the Volatile Organic Compound sensor in the form SERIAL.FUNCTIONID.

voc→get_highestValue()

Returns the maximal value observed for the estimated VOC concentration since the device was started.

voc→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

voc→get_logicalName()

Returns the logical name of the Volatile Organic Compound sensor.

voc→get_lowestValue()

Returns the minimal value observed for the estimated VOC concentration since the device was started.

voc→get_module()

Gets the YModule object for the device on which the function is located.

voc→get_module_async(**callback**, **context**)

Gets the YModule object for the device on which the function is located (asynchronous version).

voc→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

voc→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

voc→get_resolution()

Returns the resolution of the measured values.

voc→get_unit()

Returns the measuring unit for the estimated VOC concentration.

voc→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

voc→isOnline()

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error.

voc→isOnline_async(**callback**, **context**)

Checks if the Volatile Organic Compound sensor is currently reachable, without raising any error (asynchronous version).

voc→load(**msValidity**)

Preloads the Volatile Organic Compound sensor cache with a specified validity duration.

voc→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method calibrateFromPoints.

voc→load_async(**msValidity**, **callback**, **context**)

Preloads the Volatile Organic Compound sensor cache with a specified validity duration (asynchronous version).

voc→nextVoc()

Continues the enumeration of Volatile Organic Compound sensors started using yFirstVoc().

voc→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

voc→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

voc→set_highestValue(**newval**)

Changes the recorded maximal value observed.

voc→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

voc→set_logicalName(newval)

Changes the logical name of the Volatile Organic Compound sensor.

voc→set_lowestValue(newval)

Changes the recorded minimal value observed.

voc→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

voc→set_resolution(newval)

Changes the resolution of the measured physical values.

voc→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

voc→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

voc→calibrateFromPoints()YVoc calibrateFromPoints YVoc

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

YVoc **target calibrateFromPoints rawValues refValues**

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→get_advertisedValue() YVoc voc→advertisedValue()YVoc get_advertisedValue

Returns the current value of the Volatile Organic Compound sensor (no more than 6 characters).

YVoc **target** **get_advertisedValue**

Returns :

a string corresponding to the current value of the Volatile Organic Compound sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

voc→get_currentRawValue() YVoc voc→currentRawValue()YVoc get_currentRawValue

Returns the uncalibrated, unrounded raw value returned by the sensor, in ppm (vol), as a floating point number.

```
YVoc target get_currentRawValue
```

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in ppm (vol), as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

voc→get_currentValue() YVoc voc→currentValue()YVoc get_currentValue

Returns the current value of the estimated VOC concentration, in ppm (vol), as a floating point number.

```
YVoc target get_currentValue
```

Returns :

a floating point number corresponding to the current value of the estimated VOC concentration, in ppm (vol), as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

voc→get_highestValue() YVoc
voc→highestValue()YVoc get_highestValue

Returns the maximal value observed for the estimated VOC concentration since the device was started.

```
YVoc target get_highestValue
```

Returns :

a floating point number corresponding to the maximal value observed for the estimated VOC concentration since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

voc→get_logFrequency() YVoc voc→logFrequency()YVoc get_logFrequency

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

YVoc **target** **get_logFrequency**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

voc→get_logicalName() YVoc
voc→logicalName()YVoc get_logicalName

Returns the logical name of the Volatile Organic Compound sensor.

YVoc **target get_logicalName**

Returns :
a string corresponding to the logical name of the Volatile Organic Compound sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

voc→get_lowestValue() YVoc voc→lowestValue()YVoc get_lowestValue

Returns the minimal value observed for the estimated VOC concentration since the device was started.

```
YVoc target get_lowestValue
```

Returns :

a floating point number corresponding to the minimal value observed for the estimated VOC concentration since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

voc→get_recordedData() YVoc
voc→recordedData()YVoc get_recordedData

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

```
YVoc target get_recordedData startTime endTime
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

voc→get_reportFrequency() YVoc voc→reportFrequency()YVoc get_reportFrequency

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

YVoc **target** **get_reportFrequency**

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

voc→get_resolution() YVoc
voc→resolution()YVoc get_resolution

Returns the resolution of the measured values.

```
YVoc target get_resolution
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

voc→get_unit() YVoc voc→unit()YVoc get_unit

Returns the measuring unit for the estimated VOC concentration.

```
YVoc target get_unit
```

Returns :

a string corresponding to the measuring unit for the estimated VOC concentration

On failure, throws an exception or returns `Y_UNIT_INVALID`.

voc→loadCalibrationPoints()YVoc loadCalibrationPoints YVoc

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

YVoc **target loadCalibrationPoints rawValues refValues**

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→set_highestValue() YVoc voc→setHighestValue()YVoc set_highestValue

Changes the recorded maximal value observed.

```
YVoc target set_highestValue newval
```

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→set_logFrequency() YVoc
voc→setLogFrequency()YVoc set_logFrequency

Changes the datalogger recording frequency for this function.

```
YVoc target set_logFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→set_logicalName() YVoc voc→setLogicalName()YVoc set_logicalName

Changes the logical name of the Volatile Organic Compound sensor.

```
YVoc 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 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()YVoc set_lowestValue

Changes the recorded minimal value observed.

```
YVoc target set_lowestValue newval
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→set_reportFrequency() YVoc
voc→setReportFrequency()YVoc
set_reportFrequency

Changes the timed value notification frequency for this function.

```
YVoc target set_reportFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voc→set_resolution() YVoc
voc→setResolution()YVoc set_resolution

Changes the resolution of the measured physical values.

```
YVoc target set_resolution newval
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.45. Voltage function interface

The Yoctopuce application programming interface allows you to read an instant measure of the sensor, as well as the minimal and maximal values observed.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_voltage.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YVoltage = yoctolib.YVoltage;
php     require_once('yocto_voltage.php');
cpp     #include "yocto_voltage.h"
m       #import "yocto_voltage.h"
pas     uses yocto_voltage;
vb      yocto_voltage.vb
cs      yocto_voltage.cs
java    import com.yoctopuce.YoctoAPI.YVoltage;
py      from yocto_voltage import *
```

Global functions
yFindVoltage(**func**) Retrieves a voltage sensor for a given identifier.
yFirstVoltage() Starts the enumeration of voltage sensors currently accessible.
YVoltage methods
voltage→calibrateFromPoints(**rawValues**, **refValues**) Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.
voltage→describe() Returns a short text that describes unambiguously the instance of the voltage sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
voltage→get_advertisedValue() Returns the current value of the voltage sensor (no more than 6 characters).
voltage→get_currentRawValue() Returns the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number.
voltage→get_currentValue() Returns the current value of the voltage, in Volt, as a floating point number.
voltage→get_errorMessage() Returns the error message of the latest error with the voltage sensor.
voltage→get_errorType() Returns the numerical error code of the latest error with the voltage sensor.
voltage→get_friendlyName() Returns a global identifier of the voltage sensor in the format `MODULE_NAME.FUNCTION_NAME`.
voltage→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
voltage→get_functionId() Returns the hardware identifier of the voltage sensor, without reference to the module.
voltage→get_hardwareId() Returns the unique hardware identifier of the voltage sensor in the form `SERIAL.FUNCTIONID`.

voltage→get_highestValue()

Returns the maximal value observed for the voltage since the device was started.

voltage→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

voltage→get_logicalName()

Returns the logical name of the voltage sensor.

voltage→get_lowestValue()

Returns the minimal value observed for the voltage since the device was started.

voltage→get_module()

Gets the `YModule` object for the device on which the function is located.

voltage→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

voltage→get_recordedData(**startTime**, **endTime**)

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

voltage→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

voltage→get_resolution()

Returns the resolution of the measured values.

voltage→get_unit()

Returns the measuring unit for the voltage.

voltage→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

voltage→isOnline()

Checks if the voltage sensor is currently reachable, without raising any error.

voltage→isOnline_async(**callback**, **context**)

Checks if the voltage sensor is currently reachable, without raising any error (asynchronous version).

voltage→load(**msValidity**)

Preloads the voltage sensor cache with a specified validity duration.

voltage→loadCalibrationPoints(**rawValues**, **refValues**)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

voltage→load_async(**msValidity**, **callback**, **context**)

Preloads the voltage sensor cache with a specified validity duration (asynchronous version).

voltage→nextVoltage()

Continues the enumeration of voltage sensors started using `yFirstVoltage()`.

voltage→registerTimedReportCallback(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

voltage→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

voltage→set_highestValue(**newval**)

Changes the recorded maximal value observed.

voltage→set_logFrequency(**newval**)

Changes the datalogger recording frequency for this function.

voltage→set_logicalName(**newval**)

Changes the logical name of the voltage sensor.

voltage→set_lowestValue(newval)

Changes the recorded minimal value observed.

voltage→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

voltage→set_resolution(newval)

Changes the resolution of the measured physical values.

voltage→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

voltage→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

voltage→calibrateFromPoints()YVoltage calibrateFromPoints YVoltage

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

YVoltage **target calibrateFromPoints rawValues refValues**

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→get_advertisedValue() YVoltage
voltage→advertisedValue()YVoltage
get_advertisedValue

Returns the current value of the voltage sensor (no more than 6 characters).

YVoltage **target** **get_advertisedValue**

Returns :
a string corresponding to the current value of the voltage sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

voltage→get_currentRawValue() YVoltage
voltage→currentRawValue()YVoltage
get_currentRawValue

Returns the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number.

YVoltage **target get_currentRawValue**

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

voltage→get_currentValue() YVoltage
voltage→currentValue()YVoltage get_currentValue

Returns the current value of the voltage, in Volt, as a floating point number.

YVoltage **target get_currentValue**

Returns :
a floating point number corresponding to the current value of the voltage, in Volt, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

voltage→get_highestValue() YVoltage
voltage→highestValue()YVoltage get_highestValue

Returns the maximal value observed for the voltage since the device was started.

YVoltage **target get_highestValue**

Returns :

a floating point number corresponding to the maximal value observed for the voltage since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

voltage→get_logFrequency() YVoltage
voltage→logFrequency()YVoltage get_logFrequency

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

YVoltage **target** **get_logFrequency**

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns `Y_LOGFREQUENCY_INVALID`.

voltage→get_logicalName() YVoltage
voltage→logicalName()YVoltage get_logicalName

Returns the logical name of the voltage sensor.

YVoltage **target get_logicalName**

Returns :
a string corresponding to the logical name of the voltage sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

voltage→get_lowestValue() YVoltage
voltage→lowestValue()YVoltage get_lowestValue

Returns the minimal value observed for the voltage since the device was started.

YVoltage **target** **get_lowestValue**

Returns :

a floating point number corresponding to the minimal value observed for the voltage since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

voltage→get_recordedData() YVoltage
voltage→recordedData()YVoltage get_recordedData

Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.

YVoltage **target get_recordedData startTime endTime**

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the DataSet class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as DataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any meaasure, without ending limit.

Returns :

an instance of YDataSet, providing access to historical data. Past measures can be loaded progressively using methods from the YDataSet object.

voltage→get_reportFrequency() YVoltage
voltage→reportFrequency()YVoltage
get_reportFrequency

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

YVoltage **target** **get_reportFrequency**

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `Y_REPORTFREQUENCY_INVALID`.

voltage→get_resolution() YVoltage
voltage→resolution()YVoltage get_resolution

Returns the resolution of the measured values.

```
YVoltage target get_resolution
```

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

voltage→get_unit() YVoltage
voltage→unit()YVoltage get_unit

Returns the measuring unit for the voltage.

```
YVoltage target get_unit
```

Returns :

a string corresponding to the measuring unit for the voltage

On failure, throws an exception or returns `Y_UNIT_INVALID`.

voltage→loadCalibrationPoints()YVoltage loadCalibrationPoints YVoltage

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

YVoltage **target loadCalibrationPoints rawValues refValues**

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→set_highestValue() YVoltage
voltage→setHighestValue()YVoltage
set_highestValue

Changes the recorded maximal value observed.

YVoltage **target** **set_highestValue** **newval**

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→set_logFrequency() YVoltage
voltage→setLogFrequency()YVoltage
set_logFrequency

Changes the datalogger recording frequency for this function.

```
YVoltage target set_logFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→set_logicalName() YVoltage
voltage→setLogicalName()YVoltage set_logicalName

Changes the logical name of the voltage sensor.

```
YVoltage 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 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()YVoltage set_lowestValue

Changes the recorded minimal value observed.

```
YVoltage target set_lowestValue newval
```

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→set_reportFrequency() YVoltage
voltage→setReportFrequency()YVoltage
set_reportFrequency

Changes the timed value notification frequency for this function.

```
YVoltage target set_reportFrequency newval
```

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→set_resolution() YVoltage
voltage→setResolution()YVoltage set_resolution

Changes the resolution of the measured physical values.

```
YVoltage target set_resolution newval
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.46. Voltage source function interface

Yoctopuce application programming interface allows you to control the module voltage output. You affect absolute output values or make transitions

In order to use the functions described here, you should include:

```
js    <script type='text/javascript' src='yocto_vsource.js'></script>
php   require_once('yocto_vsource.php');
cpp   #include "yocto_vsource.h"
m     #import "yocto_vsource.h"
pas   uses yocto_vsource;
vb    yocto_vsource.vb
cs    yocto_vsource.cs
java  import com.yoctopuce.YoctoAPI.YVSource;
py    from yocto_vsource import *
```

Global functions
yFindVSource(**func**) Retrieves a voltage source for a given identifier.
yFirstVSource() Starts the enumeration of voltage sources currently accessible.
YVSource methods
vsource→describe() Returns a short text that describes the function in the form TYPE(NAME)=SERIAL.FUNCTIONID.
vsource→get_advertisedValue() Returns the current value of the voltage source (no more than 6 characters).
vsource→get_errorMessage() Returns the error message of the latest error with this function.
vsource→get_errorType() Returns the numerical error code of the latest error with this function.
vsource→get_extPowerFailure() Returns true if external power supply voltage is too low.
vsource→get_failure() Returns true if the module is in failure mode.
vsource→get_friendlyName() Returns a global identifier of the function in the format MODULE_NAME.FUNCTION_NAME.
vsource→get_functionDescriptor() Returns a unique identifier of type YFUN_DESCR corresponding to the function.
vsource→get_functionId() Returns the hardware identifier of the function, without reference to the module.
vsource→get_hardwareId() Returns the unique hardware identifier of the function in the form SERIAL.FUNCTIONID.
vsource→get_logicalName() Returns the logical name of the voltage source.
vsource→get_module() Gets the YModule object for the device on which the function is located.
vsource→get_module_async(**callback**, **context**)

Gets the YModule object for the device on which the function is located (asynchronous version).

vsource→get_overCurrent()

Returns true if the appliance connected to the device is too greedy .

vsource→get_overHeat()

Returns TRUE if the module is overheating.

vsource→get_overLoad()

Returns true if the device is not able to maintaint the requested voltage output .

vsource→get_regulationFailure()

Returns true if the voltage output is too high regarding the requested voltage .

vsource→get_unit()

Returns the measuring unit for the voltage.

vsource→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

vsource→get_voltage()

Returns the voltage output command (mV)

vsource→isOnline()

Checks if the function is currently reachable, without raising any error.

vsource→isOnline_async(**callback**, **context**)

Checks if the function is currently reachable, without raising any error (asynchronous version).

vsource→load(**msValidity**)

Preloads the function cache with a specified validity duration.

vsource→load_async(**msValidity**, **callback**, **context**)

Preloads the function cache with a specified validity duration (asynchronous version).

vsource→nextVSource()

Continues the enumeration of voltage sources started using yFirstVSource().

vsource→pulse(**voltage**, **ms_duration**)

Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.

vsource→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

vsource→set_logicalName(**newval**)

Changes the logical name of the voltage source.

vsource→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

vsource→set_voltage(**newval**)

Tunes the device output voltage (milliVolts).

vsource→voltageMove(**target**, **ms_duration**)

Performs a smooth move at constant speed toward a given value.

vsource→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

vsource→get_advertisedValue() YVSource
vsource→advertisedValue()YVSource
get_advertisedValue

Returns the current value of the voltage source (no more than 6 characters).

YVSource **target** **get_advertisedValue**

Returns :
a string corresponding to the current value of the voltage source (no more than 6 characters)

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

vsource→get_extPowerFailure() YVSource
vsource→extPowerFailure()YVSource
get_extPowerFailure

Returns true if external power supply voltage is too low.

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()YVSource get_failure

Returns true if the module is in failure mode.

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_logicalName() YVSource
vsource→logicalName()YVSource get_logicalName

Returns the logical name of the voltage source.

YVSource **target get_logicalName**

Returns :
a string corresponding to the logical name of the voltage source

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

vsource→get_overCurrent() YVSource
vsource→overCurrent()YVSource get_overCurrent

Returns true if the appliance connected to the device is too greedy .

YVSource **target get_overCurrent**

Returns :

either `Y_OVERCURRENT_FALSE` or `Y_OVERCURRENT_TRUE`, according to true if the appliance connected to the device is too greedy

On failure, throws an exception or returns `Y_OVERCURRENT_INVALID`.

vsource→get_overHeat() YVSource
vsource→overHeat()YVSource get_overHeat

Returns TRUE if the module is overheating.

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.

vsource→get_overLoad() YVSource
vsource→overLoad()YVSource get_overLoad

Returns true if the device is not able to maintaint the requested voltage output .

YVSource **target** **get_overLoad**

Returns :

either `Y_OVERLOAD_FALSE` or `Y_OVERLOAD_TRUE`, according to true if the device is not able to maintaint the requested voltage output

On failure, throws an exception or returns `Y_OVERLOAD_INVALID`.

vsource→get_regulationFailure() YVSource
vsource→regulationFailure()YVSource
get_regulationFailure

Returns true if the voltage output is too high regarding the requested voltage .

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() YVSource
vsource→unit()YVSource get_unit

Returns the measuring unit for the voltage.

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→pulse()YVSource pulse — YVSource

Sets device output to a specific volatage, for a specified duration, then brings it automatically to 0V.

YVSource **target pulse voltage ms_duration**

Parameters :

voltage	pulse voltage, in millivolts
ms_duration	pulse duration, in millisecondes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

vsource→set_logicalName() YVSource
vsource→setLogicalName()YVSource
set_logicalName

Changes the logical name of the voltage source.

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_voltage() YVSource
vsource→setVoltage()YVSource set_voltage

Tunes the device output voltage (milliVolts).

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()YVSource voltageMove YVSource

Performs a smooth move at constant speed toward a given value.

YVSource **target voltageMove target ms_duration**

Parameters :

target	new output value at end of transition, in milliVolts.
ms_duration	transition duration, in milliseconds

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.47. WakeUpMonitor function interface

The WakeUpMonitor function handles globally all wake-up sources, as well as automated sleep mode.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_wakeupmonitor.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YWakeUpMonitor = yoctolib.YWakeUpMonitor;
php     require_once('yocto_wakeupmonitor.php');
cpp     #include "yocto_wakeupmonitor.h"
m       #import "yocto_wakeupmonitor.h"
pas     uses yocto_wakeupmonitor;
vb      yocto_wakeupmonitor.vb
cs      yocto_wakeupmonitor.cs
java    import com.yoctopuce.YoctoAPI.YWakeUpMonitor;
py      from yocto_wakeupmonitor import *
```

Global functions
yFindWakeUpMonitor(**func**) Retrieves a monitor for a given identifier.
yFirstWakeUpMonitor() Starts the enumeration of monitors currently accessible.
`YWakeUpMonitor` methods
wakeupmonitor→describe() Returns a short text that describes unambiguously the instance of the monitor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
wakeupmonitor→get_advertisedValue() Returns the current value of the monitor (no more than 6 characters).
wakeupmonitor→get_errorMessage() Returns the error message of the latest error with the monitor.
wakeupmonitor→get_errorType() Returns the numerical error code of the latest error with the monitor.
wakeupmonitor→get_friendlyName() Returns a global identifier of the monitor in the format `MODULE_NAME.FUNCTION_NAME`.
wakeupmonitor→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
wakeupmonitor→get_functionId() Returns the hardware identifier of the monitor, without reference to the module.
wakeupmonitor→get_hardwareId() Returns the unique hardware identifier of the monitor in the form `SERIAL.FUNCTIONID`.
wakeupmonitor→get_logicalName() Returns the logical name of the monitor.
wakeupmonitor→get_module() Gets the `YModule` object for the device on which the function is located.
wakeupmonitor→get_module_async(**callback**, **context**) Gets the `YModule` object for the device on which the function is located (asynchronous version).
wakeupmonitor→get_nextWakeUp()

Returns the next scheduled wake up date/time (UNIX format)

wakeupmonitor→get_powerDuration()

Returns the maximal wake up time (in seconds) before automatically going to sleep.

wakeupmonitor→get_sleepCountdown()

Returns the delay before the next sleep period.

wakeupmonitor→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

wakeupmonitor→get_wakeUpReason()

Returns the latest wake up reason.

wakeupmonitor→get_wakeUpState()

Returns the current state of the monitor

wakeupmonitor→isOnline()

Checks if the monitor is currently reachable, without raising any error.

wakeupmonitor→isOnline_async(**callback**, **context**)

Checks if the monitor is currently reachable, without raising any error (asynchronous version).

wakeupmonitor→load(**msValidity**)

Preloads the monitor cache with a specified validity duration.

wakeupmonitor→load_async(**msValidity**, **callback**, **context**)

Preloads the monitor cache with a specified validity duration (asynchronous version).

wakeupmonitor→nextWakeUpMonitor()

Continues the enumeration of monitors started using `yFirstWakeUpMonitor()`.

wakeupmonitor→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

wakeupmonitor→resetSleepCountDown()

Resets the sleep countdown.

wakeupmonitor→set_logicalName(**newval**)

Changes the logical name of the monitor.

wakeupmonitor→set_nextWakeUp(**newval**)

Changes the days of the week when a wake up must take place.

wakeupmonitor→set_powerDuration(**newval**)

Changes the maximal wake up time (seconds) before automatically going to sleep.

wakeupmonitor→set_sleepCountdown(**newval**)

Changes the delay before the next sleep period.

wakeupmonitor→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

wakeupmonitor→sleep(**secBeforeSleep**)

Goes to sleep until the next wake up condition is met, the RTC time must have been set before calling this function.

wakeupmonitor→sleepFor(**secUntilWakeUp**, **secBeforeSleep**)

Goes to sleep for a specific duration or until the next wake up condition is met, the RTC time must have been set before calling this function.

wakeupmonitor→sleepUntil(**wakeUpTime**, **secBeforeSleep**)

Go to sleep until a specific date is reached or until the next wake up condition is met, the RTC time must have been set before calling this function.

wakeupmonitor→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

wakeupmonitor→wakeUp()

Forces a wake up.

wakeupmonitor→get_advertisedValue() YWakeUpMonitor wakeupmonitor→advertisedValue()YWakeUpMonitor get_advertisedValue

Returns the current value of the monitor (no more than 6 characters).

YWakeUpMonitor **target** **get_advertisedValue**

Returns :
a string corresponding to the current value of the monitor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

wakeupmonitor→get_logicalName() YWakeUpMonitor
wakeupmonitor→logicalName()YWakeUpMonitor
get_logicalName

Returns the logical name of the monitor.

YWakeUpMonitor **target get_logicalName**

Returns :
a string corresponding to the logical name of the monitor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

wakeupmonitor→get_powerDuration() YWakeUpMonitor wakeupmonitor→powerDuration()YWakeUpMonitor get_powerDuration

Returns the maximal wake up time (in seconds) before automatically going to sleep.

YWakeUpMonitor **target** **get_powerDuration**

Returns :
an integer corresponding to the maximal wake up time (in seconds) before automatically going to sleep

On failure, throws an exception or returns `Y_POWERDURATION_INVALID`.

wakeupmonitor→get_sleepCountdown() YWakeUpMonitor
wakeupmonitor→sleepCountdown()YWakeUpMonitor
get_sleepCountdown

Returns the delay before the next sleep period.

YWakeUpMonitor **target get_sleepCountdown**

Returns :
an integer corresponding to the delay before the next sleep period

On failure, throws an exception or returns `Y_SLEEPCOUNTDOWN_INVALID`.

wakeupmonitor→get_wakeUpReason() YWakeUpMonitor wakeupmonitor→wakeUpReason()YWakeUpMonitor get_wakeUpReason

Returns the latest wake up reason.

YWakeUpMonitor **target** **get_wakeUpReason**

Returns :

a value among `Y_WAKEUPREASON_USBPOWER`, `Y_WAKEUPREASON_EXTPOWER`, `Y_WAKEUPREASON_ENDOFSLEEP`, `Y_WAKEUPREASON_EXTSIG1`, `Y_WAKEUPREASON_SCHEDULE1` and `Y_WAKEUPREASON_SCHEDULE2` corresponding to the latest wake up reason

On failure, throws an exception or returns `Y_WAKEUPREASON_INVALID`.

wakeupmonitor→resetSleepCountDown() YWakeUpMonitor
YWakeUpMonitor resetSleepCountDown

Resets the sleep countdown.

YWakeUpMonitor **target resetSleepCountDown**

Returns :

YAPI_SUCCESS if the call succeeds. On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_logicalName() YWakeUpMonitor wakeupmonitor→setLogicalName()YWakeUpMonitor set_logicalName

Changes the logical name of the monitor.

```
YWakeUpMonitor 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 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()YWakeUpMonitor set_nextWakeUp

Changes the days of the week when a wake up must take place.

YWakeUpMonitor **target set_nextWakeUp newval**

Parameters :

newval an integer corresponding to the days of the week when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_powerDuration() YWakeUpMonitor
wakeupmonitor→setPowerDuration()
YWakeUpMonitor set_powerDuration

Changes the maximal wake up time (seconds) before automatically going to sleep.

YWakeUpMonitor **target set_powerDuration newval**

Parameters :

newval an integer corresponding to the maximal wake up time (seconds) before automatically going to sleep

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→set_sleepCountdown() YWakeUpMonitor
wakeupmonitor→setSleepCountdown()
YWakeUpMonitor set_sleepCountdown

Changes the delay before the next sleep period.

```
YWakeUpMonitor target set_sleepCountdown newval
```

Parameters :

newval an integer corresponding to the delay before the next sleep period

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→sleep()YWakeUpMonitor sleep YWakeUpMonitor

Goes to sleep until the next wake up condition is met, the RTC time must have been set before calling this function.

YWakeUpMonitor **target** **sleep** **secBeforeSleep**

Parameters :

secBeforeSleep number of seconds before going into sleep mode,

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→sleepFor()YWakeUpMonitor 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.

YWakeUpMonitor **target sleepFor secUntilWakeUp secBeforeSleep**

The count down before sleep can be canceled with resetSleepCountDown.

Parameters :

secUntilWakeUp number of seconds before next wake up

secBeforeSleep number of seconds before going into sleep mode

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→sleepUntil()YWakeUpMonitor 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.

YWakeUpMonitor **target sleepUntil wakeUpTime secBeforeSleep**

The count down before sleep can be canceled with resetSleepCountDown.

Parameters :

wakeUpTime	wake-up datetime (UNIX format)
secBeforeSleep	number of seconds before going into sleep mode

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupmonitor→wakeUp()YWakeUpMonitor wakeUp YWakeUpMonitor

Forces a wake up.

YWakeUpMonitor **target wakeUp**

3.48. WakeUpSchedule function interface

The WakeUpSchedule function implements a wake up condition. The wake up time is specified as a set of months and/or days and/or hours and/or minutes when the wake up should happen.

In order to use the functions described here, you should include:

```
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');
cpp     #include "yocto_wakeupschedule.h"
m       #import "yocto_wakeupschedule.h"
pas     uses yocto_wakeupschedule;
vb      yocto_wakeupschedule.vb
cs      yocto_wakeupschedule.cs
java    import com.yoctopuce.YoctoAPI.YWakeUpSchedule;
py      from yocto_wakeupschedule import *
```

Global functions

yFindWakeUpSchedule(**func**)

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 intervalof each hour scheduled for wake up.

wakeupschedule→get_module()
Gets the `YModule` object for the device on which the function is located.

wakeupschedule→get_module_async(**callback**, **context**)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

wakeupschedule→get_monthDays()
Returns the days of the month scheduled for wake up.

wakeupschedule→get_months()
Returns the months scheduled for wake up.

wakeupschedule→get_nextOccurence()
Returns the date/time (seconds) of the next wake up occurence

wakeupschedule→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

wakeupschedule→get_weekDays()
Returns the days of the week scheduled for wake up.

wakeupschedule→isOnline()
Checks if the wake up schedule is currently reachable, without raising any error.

wakeupschedule→isOnline_async(**callback**, **context**)
Checks if the wake up schedule is currently reachable, without raising any error (asynchronous version).

wakeupschedule→load(**msValidity**)
Preloads the wake up schedule cache with a specified validity duration.

wakeupschedule→load_async(**msValidity**, **callback**, **context**)
Preloads the wake up schedule cache with a specified validity duration (asynchronous version).

wakeupschedule→nextWakeUpSchedule()
Continues the enumeration of wake up schedules started using `yFirstWakeUpSchedule()`.

wakeupschedule→registerValueCallback(**callback**)
Registers the callback function that is invoked on every change of advertised value.

wakeupschedule→set_hours(**newval**)
Changes the hours when a wake up must take place.

wakeupschedule→set_logicalName(**newval**)
Changes the logical name of the wake up schedule.

wakeupschedule→set_minutes(**bitmap**)
Changes all the minutes where a wake up must take place.

wakeupschedule→set_minutesA(**newval**)
Changes the minutes in the 00-29 interval when a wake up must take place.

wakeupschedule→set_minutesB(**newval**)
Changes the minutes in the 30-59 interval when a wake up must take place.

wakeupschedule→set_monthDays(**newval**)
Changes the days of the month when a wake up must take place.

wakeupschedule→set_months(**newval**)
Changes the months when a wake up must take place.

wakeupschedule→set_userData(**data**)
Stores a user context provided as argument in the userData attribute of the function.

wakeupschedule→set_weekDays(newval)

Changes the days of the week when a wake up must take place.

wakeupschedule→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

wakeupschedule→get_advertisedValue() YWakeUpSchedule
wakeupschedule→advertisedValue()
YWakeUpSchedule get_advertisedValue

Returns the current value of the wake up schedule (no more than 6 characters).

```
YWakeUpSchedule target get_advertisedValue
```

Returns :
a string corresponding to the current value of the wake up schedule (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

wakeupschedule→get_hours() YWakeUpSchedule
wakeupschedule→hours()YWakeUpSchedule
get_hours

Returns the hours scheduled for wake up.

```
YWakeUpSchedule target get_hours
```

Returns :
an integer corresponding to the hours scheduled for wake up

On failure, throws an exception or returns `Y_HOURS_INVALID`.

wakeupschedule→get_logicalName() YWakeUpSchedule
wakeupschedule→logicalName()YWakeUpSchedule
get_logicalName

Returns the logical name of the wake up schedule.

```
YWakeUpSchedule target get_logicalName
```

Returns :
a string corresponding to the logical name of the wake up schedule.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

wakeupschedule→get_minutes() YWakeUpSchedule
wakeupschedule→minutes()YWakeUpSchedule
get_minutes

Returns all the minutes of each hour that are scheduled for wake up.

```
YWakeUpSchedule target get_minutes
```

wakeupschedule→get_minutesA() YWakeUpSchedule
wakeupschedule→minutesA()YWakeUpSchedule
get_minutesA

Returns the minutes in the 00-29 interval of each hour scheduled for wake up.

YWakeUpSchedule **target** **get_minutesA**

Returns :
an integer corresponding to the minutes in the 00-29 interval of each hour scheduled for wake up

On failure, throws an exception or returns `Y_MINUTESA_INVALID`.

wakeupschedule→get_minutesB() YWakeUpSchedule wakeupschedule→minutesB()YWakeUpSchedule get_minutesB

Returns the minutes in the 30-59 intervalof each hour scheduled for wake up.

YWakeUpSchedule **target get_minutesB**

Returns :
an integer corresponding to the minutes in the 30-59 intervalof each hour scheduled for wake up

On failure, throws an exception or returns `Y_MINUTESB_INVALID`.

wakeupschedule→get_monthDays() YWakeUpSchedule
wakeupschedule→monthDays()YWakeUpSchedule
get_monthDays

Returns the days of the month scheduled for wake up.

YWakeUpSchedule **target** **get_monthDays**

Returns :
an integer corresponding to the days of the month scheduled for wake up

On failure, throws an exception or returns `Y_MONTHDAYS_INVALID`.

wakeupschedule→get_months() YWakeUpSchedule
wakeupschedule→months()YWakeUpSchedule
get_months

Returns the months scheduled for wake up.

```
YWakeUpSchedule target get_months
```

Returns :
an integer corresponding to the months scheduled for wake up

On failure, throws an exception or returns `Y_MONTHS_INVALID`.

wakeupschedule→get_weekDays() YWakeUpSchedule
wakeupschedule→weekDays()YWakeUpSchedule
get_weekDays

Returns the days of the week scheduled for wake up.

YWakeUpSchedule **target** **get_weekDays**

Returns :
an integer corresponding to the days of the week scheduled for wake up

On failure, throws an exception or returns `Y_WEEKDAYS_INVALID`.

wakeupschedule→set_hours() YWakeUpSchedule
wakeupschedule→setHours()YWakeUpSchedule
set_hours

Changes the hours when a wake up must take place.

```
YWakeUpSchedule target set_hours newval
```

Parameters :

newval an integer corresponding to the hours when a wake up must take place

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_logicalName() YWakeUpSchedule
wakeupschedule→setLogicalName()
YWakeUpSchedule set_logicalName

Changes the logical name of the wake up schedule.

```
YWakeUpSchedule 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 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()YWakeUpSchedule
set_minutes

Changes all the minutes where a wake up must take place.

YWakeUpSchedule **target set_minutes bitmap**

Parameters :

bitmap Minutes 00-59 of each hour scheduled for wake up.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_minutesA() YWakeUpSchedule
wakeupschedule→setMinutesA()YWakeUpSchedule
set_minutesA

Changes the minutes in the 00-29 interval when a wake up must take place.

YWakeUpSchedule **target set_minutesA newval**

Parameters :

newval an integer corresponding to the minutes in the 00-29 interval when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_minutesB() YWakeUpSchedule
wakeupschedule→setMinutesB()YWakeUpSchedule
set_minutesB

Changes the minutes in the 30-59 interval when a wake up must take place.

YWakeUpSchedule **target set_minutesB newval**

Parameters :

newval an integer corresponding to the minutes in the 30-59 interval when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_monthDays() YWakeUpSchedule
wakeupschedule→setMonthDays()
YWakeUpSchedule set_monthDays

Changes the days of the month when a wake up must take place.

```
YWakeUpSchedule target set_monthDays newval
```

Parameters :

newval an integer corresponding to the days of the month when a wake up must take place

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_months() YWakeUpSchedule
wakeupschedule→setMonths()YWakeUpSchedule
set_months

Changes the months when a wake up must take place.

```
YWakeUpSchedule target set_months newval
```

Parameters :

newval an integer corresponding to the months when a wake up must take place

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

wakeupschedule→set_weekDays() YWakeUpSchedule
wakeupschedule→setWeekDays()YWakeUpSchedule
set_weekDays

Changes the days of the week when a wake up must take place.

```
YWakeUpSchedule target set_weekDays newval
```

Parameters :

newval an integer corresponding to the days of the week when a wake up must take place

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.49. Watchdog function interface

The watchog function works like a relay and can cause a brief power cut to an appliance after a preset delay to force this appliance to reset. The Watchdog must be called from time to time to reset the timer and prevent the appliance reset. The watchdog can be driven direcly with *pulse* and *delayedpulse* methods to switch off an appliance for a given duration.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_watchdog.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YWatchdog = yoctolib.YWatchdog;
php     require_once('yocto_watchdog.php');
cpp     #include "yocto_watchdog.h"
m       #import "yocto_watchdog.h"
pas     uses yocto_watchdog;
vb      yocto_watchdog.vb
cs      yocto_watchdog.cs
java    import com.yoctopuce.YoctoAPI.YWatchdog;
py      from yocto_watchdog import *
```

Global functions
yFindWatchdog(**func**) Retrieves a watchdog for a given identifier.
yFirstWatchdog() Starts the enumeration of watchdog currently accessible.
`YWatchdog` methods
watchdog→delayedPulse(**ms_delay**, **ms_duration**) Schedules a pulse.
watchdog→describe() Returns a short text that describes unambiguously the instance of the watchdog in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.
watchdog→get_advertisedValue() Returns the current value of the watchdog (no more than 6 characters).
watchdog→get_autoStart() Returns the watchdog runing state at module power on.
watchdog→get_countdown() Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.
watchdog→get_errorMessage() Returns the error message of the latest error with the watchdog.
watchdog→get_errorType() Returns the numerical error code of the latest error with the watchdog.
watchdog→get_friendlyName() Returns a global identifier of the watchdog in the format `MODULE_NAME.FUNCTION_NAME`.
watchdog→get_functionDescriptor() Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.
watchdog→get_functionId() Returns the hardware identifier of the watchdog, without reference to the module.

watchdog→get_hardwareId()

Returns the unique hardware identifier of the watchdog in the form SERIAL.FUNCTIONID.

watchdog→get_logicalName()

Returns the logical name of the watchdog.

watchdog→get_maxTimeOnStateA()

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

watchdog→get_maxTimeOnStateB()

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

watchdog→get_module()

Gets the YModule object for the device on which the function is located.

watchdog→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

watchdog→get_output()

Returns the output state of the watchdog, when used as a simple switch (single throw).

watchdog→get_pulseTimer()

Returns the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation.

watchdog→get_running()

Returns the watchdog running state.

watchdog→get_state()

Returns the state of the watchdog (A for the idle position, B for the active position).

watchdog→get_stateAtPowerOn()

Returns the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change).

watchdog→get_triggerDelay()

Returns the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds.

watchdog→get_triggerDuration()

Returns the duration of resets caused by the watchdog, in milliseconds.

watchdog→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

watchdog→isOnline()

Checks if the watchdog is currently reachable, without raising any error.

watchdog→isOnline_async(callback, context)

Checks if the watchdog is currently reachable, without raising any error (asynchronous version).

watchdog→load(msValidity)

Preloads the watchdog cache with a specified validity duration.

watchdog→load_async(msValidity, callback, context)

Preloads the watchdog cache with a specified validity duration (asynchronous version).

watchdog→nextWatchdog()

Continues the enumeration of watchdog started using yFirstWatchdog().

watchdog→pulse(ms_duration)

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

watchdog→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

watchdog→resetWatchdog()

Resets the watchdog.

watchdog→set_autoStart(newval)

Changes the watchdog runningsttae at module power on.

watchdog→set_logicalName(newval)

Changes the logical name of the watchdog.

watchdog→set_maxTimeOnStateA(newval)

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

watchdog→set_maxTimeOnStateB(newval)

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

watchdog→set_output(newval)

Changes the output state of the watchdog, when used as a simple switch (single throw).

watchdog→set_running(newval)

Changes the running state of the watchdog.

watchdog→set_state(newval)

Changes the state of the watchdog (A for the idle position, B for the active position).

watchdog→set_stateAtPowerOn(newval)

Preset the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no modification).

watchdog→set_triggerDelay(newval)

Changes the waiting delay before a reset is triggered by the watchdog, in milliseconds.

watchdog→set_triggerDuration(newval)

Changes the duration of resets caused by the watchdog, in milliseconds.

watchdog→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

watchdog→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

watchdog→delayedPulse()YWatchdog delayedPulse YWatchdog

Schedules a pulse.

YWatchdog **target delayedPulse ms_delay ms_duration**

Parameters :

ms_delay waiting time before the pulse, in millisecondes

ms_duration pulse duration, in millisecondes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→get_advertisedValue() YWatchdog
watchdog→advertisedValue()YWatchdog
get_advertisedValue

Returns the current value of the watchdog (no more than 6 characters).

YWatchdog **target** **get_advertisedValue**

Returns :
a string corresponding to the current value of the watchdog (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

watchdog→get_autoStart() YWatchdog
watchdog→autoStart()YWatchdog get_autoStart

Returns the watchdog runing state at module power on.

YWatchdog **target** **get_autoStart**

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()YWatchdog get_countdown

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

YWatchdog **target** **get_countdown**

Returns :

an integer corresponding to the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero

On failure, throws an exception or returns `Y_COUNTDOWN_INVALID`.

watchdog→get_logicalName() YWatchdog
watchdog→logicalName()YWatchdog
get_logicalName

Returns the logical name of the watchdog.

YWatchdog **target get_logicalName**

Returns :
a string corresponding to the logical name of the watchdog.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

watchdog→get_maxTimeOnStateA() YWatchdog
watchdog→maxTimeOnStateA()YWatchdog
get_maxTimeOnStateA

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

YWatchdog **target** **get_maxTimeOnStateA**

Zero means no maximum time.

Returns :
an integer

On failure, throws an exception or returns `Y_MAXTIMEONSTATEA_INVALID`.

watchdog→get_maxTimeOnStateB() YWatchdog
watchdog→maxTimeOnStateB()YWatchdog
get_maxTimeOnStateB

Retourne the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

YWatchdog **target get_maxTimeOnStateB**

Zero means no maximum time.

Returns :
an integer

On failure, throws an exception or returns `Y_MAXTIMEONSTATEB_INVALID`.

watchdog→get_output() YWatchdog
watchdog→output()YWatchdog get_output

Returns the output state of the watchdog, when used as a simple switch (single throw).

YWatchdog **target** **get_output**

Returns :

either `Y_OUTPUT_OFF` or `Y_OUTPUT_ON`, according to the output state of the watchdog, when used as a simple switch (single throw)

On failure, throws an exception or returns `Y_OUTPUT_INVALID`.

watchdog→get_pulseTimer() YWatchdog
watchdog→pulseTimer()YWatchdog get_pulseTimer

Returns the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation.

```
YWatchdog target get_pulseTimer
```

When there is no ongoing pulse, returns zero.

Returns :

an integer corresponding to the number of milliseconds remaining before the watchdog is returned to idle position (state A), during a measured pulse generation

On failure, throws an exception or returns `Y_PULSETIMER_INVALID`.

watchdog→get_running() YWatchdog
watchdog→running()YWatchdog get_running

Returns the watchdog running state.

YWatchdog **target** **get_running**

Returns :

either `Y_RUNNING_OFF` or `Y_RUNNING_ON`, according to the watchdog running state

On failure, throws an exception or returns `Y_RUNNING_INVALID`.

watchdog→get_state() YWatchdog
watchdog→state()YWatchdog get_state

Returns the state of the watchdog (A for the idle position, B for the active position).

YWatchdog **target** **get_state**

Returns :

either Y_STATE_A or Y_STATE_B, according to the state of the watchdog (A for the idle position, B for the active position)

On failure, throws an exception or returns Y_STATE_INVALID.

watchdog→get_stateAtPowerOn() YWatchdog
watchdog→stateAtPowerOn()YWatchdog
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).

YWatchdog **target** **get_stateAtPowerOn**

Returns :

a value among `Y_STATEATPOWERON_UNCHANGED`, `Y_STATEATPOWERON_A` and `Y_STATEATPOWERON_B` corresponding to the state of the watchdog at device startup (A for the idle position, B for the active position, UNCHANGED for no change)

On failure, throws an exception or returns `Y_STATEATPOWERON_INVALID`.

watchdog→get_triggerDelay() YWatchdog
watchdog→triggerDelay()YWatchdog
get_triggerDelay

Returns the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds.

YWatchdog **target** **get_triggerDelay**

Returns :

an integer corresponding to the waiting duration before a reset is automatically triggered by the watchdog, in milliseconds

On failure, throws an exception or returns `Y_TRIGGERDELAY_INVALID`.

watchdog→get_triggerDuration() YWatchdog
watchdog→triggerDuration()YWatchdog
get_triggerDuration

Returns the duration of resets caused by the watchdog, in milliseconds.

YWatchdog **target** **get_triggerDuration**

Returns :
an integer corresponding to the duration of resets caused by the watchdog, in milliseconds

On failure, throws an exception or returns `Y_TRIGGERDURATION_INVALID`.

watchdog→pulse()YWatchdog pulse YWatchdog

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

YWatchdog **target pulse ms_duration**

Parameters :

ms_duration pulse duration, in millisecondes

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→resetWatchdog()YWatchdog YWatchdog resetWatchdog

Resets the watchdog.

YWatchdog **target** **resetWatchdog**

When the watchdog is running, this function must be called on a regular basis to prevent the watchog 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()YWatchdog set_autoStart

Changes the watchdog runningsttae at module power on.

YWatchdog **target** **set_autoStart** **newval**

Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

newval either `Y_AUTOSTART_OFF` or `Y_AUTOSTART_ON`, according to the watchdog runningsttae 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()YWatchdog
set_logicalName

Changes the logical name of the watchdog.

```
YWatchdog 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 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()YWatchdog
set_maxTimeOnStateA

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state A before automatically switching back in to B state.

YWatchdog **target set_maxTimeOnStateA newval**

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_maxTimeOnStateB() YWatchdog
watchdog→setMaxTimeOnStateB()YWatchdog
set_maxTimeOnStateB

Sets the maximum time (ms) allowed for $THEFUNCTIONS$ to stay in state B before automatically switching back in to A state.

```
YWatchdog target set_maxTimeOnStateB newval
```

Use zero for no maximum time.

Parameters :

newval an integer

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_output() YWatchdog
watchdog→setOutput()YWatchdog set_output

Changes the output state of the watchdog, when used as a simple switch (single throw).

YWatchdog **target set_output newval**

Parameters :

newval either Y_OUTPUT_OFF or Y_OUTPUT_ON, according to the output state of the watchdog, when used as a simple switch (single throw)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_running() YWatchdog
watchdog→setRunning()YWatchdog set_running

Changes the running state of the watchdog.

YWatchdog **target** **set_running** **newval**

Parameters :

newval either `Y_RUNNING_OFF` or `Y_RUNNING_ON`, according to the running state of the watchdog

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_state() YWatchdog
watchdog→setState()YWatchdog set_state

Changes the state of the watchdog (A for the idle position, B for the active position).

YWatchdog **target set_state newval**

Parameters :

newval either Y_STATE_A or Y_STATE_B, according to the state of the watchdog (A for the idle position, B for the active position)

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_stateAtPowerOn() YWatchdog
watchdog→setStateAtPowerOn()YWatchdog
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).

```
YWatchdog target set_stateAtPowerOn newval
```

Remember to call the matching module `saveToFlash()` method, otherwise this call will have no effect.

Parameters :

newval a value among `Y_STATEATPOWERON_UNCHANGED`, `Y_STATEATPOWERON_A` and `Y_STATEATPOWERON_B`

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_triggerDelay() YWatchdog
watchdog→setTriggerDelay()YWatchdog
set_triggerDelay

Changes the waiting delay before a reset is triggered by the watchdog, in milliseconds.

YWatchdog **target set_triggerDelay newval**

Parameters :

newval an integer corresponding to the waiting delay before a reset is triggered by the watchdog, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

watchdog→set_triggerDuration() YWatchdog
watchdog→setTriggerDuration()YWatchdog
set_triggerDuration

Changes the duration of resets caused by the watchdog, in milliseconds.

YWatchdog **target** **set_triggerDuration** **newval**

Parameters :

newval an integer corresponding to the duration of resets caused by the watchdog, in milliseconds

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

3.50. Wireless function interface

YWireless functions provides control over wireless network parameters and status for devices that are wireless-enabled.

In order to use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_wireless.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YWireless = yoctolib.YWireless;
php     require_once('yocto_wireless.php');
cpp     #include "yocto_wireless.h"
m       #import "yocto_wireless.h"
pas     uses yocto_wireless;
vb      yocto_wireless.vb
cs      yocto_wireless.cs
java    import com.yoctopuce.YoctoAPI.YWireless;
py      from yocto_wireless import *
```

Global functions

yFindWireless(**func**)

Retrieves a wireless lan interface for a given identifier.

yFirstWireless()

Starts the enumeration of wireless lan interfaces currently accessible.

`YWireless` methods

wireless→adhocNetwork(**ssid**, **securityKey**)

Changes the configuration of the wireless lan interface to create an ad-hoc wireless network, without using an access point.

wireless→describe()

Returns a short text that describes unambiguously the instance of the wireless lan interface in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

wireless→get_advertisedValue()

Returns the current value of the wireless lan interface (no more than 6 characters).

wireless→get_channel()

Returns the 802.11 channel currently used, or 0 when the selected network has not been found.

wireless→get_detectedWlans()

Returns a list of YWlanRecord objects that describe detected Wireless networks.

wireless→get_errorMessage()

Returns the error message of the latest error with the wireless lan interface.

wireless→get_errorType()

Returns the numerical error code of the latest error with the wireless lan interface.

wireless→get_friendlyName()

Returns a global identifier of the wireless lan interface in the format `MODULE_NAME.FUNCTION_NAME`.

wireless→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

wireless→get_functionId()

Returns the hardware identifier of the wireless lan interface, without reference to the module.

wireless→get_hardwareId()

Returns the unique hardware identifier of the wireless lan interface in the form `SERIAL.FUNCTIONID`.

wireless→get_linkQuality()

Returns the link quality, expressed in percent.

wireless→get_logicalName()

Returns the logical name of the wireless lan interface.

wireless→get_message()

Returns the latest status message from the wireless interface.

wireless→get_module()

Gets the `YModule` object for the device on which the function is located.

wireless→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

wireless→get_security()

Returns the security algorithm used by the selected wireless network.

wireless→get_ssid()

Returns the wireless network name (SSID).

wireless→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

wireless→isOnline()

Checks if the wireless lan interface is currently reachable, without raising any error.

wireless→isOnline_async(**callback**, **context**)

Checks if the wireless lan interface is currently reachable, without raising any error (asynchronous version).

wireless→joinNetwork(**ssid**, **securityKey**)

Changes the configuration of the wireless lan interface to connect to an existing access point (infrastructure mode).

wireless→load(**msValidity**)

Preloads the wireless lan interface cache with a specified validity duration.

wireless→load_async(**msValidity**, **callback**, **context**)

Preloads the wireless lan interface cache with a specified validity duration (asynchronous version).

wireless→nextWireless()

Continues the enumeration of wireless lan interfaces started using `yFirstWireless()`.

wireless→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

wireless→set_logicalName(**newval**)

Changes the logical name of the wireless lan interface.

wireless→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

wireless→softAPNetwork(**ssid**, **securityKey**)

Changes the configuration of the wireless lan interface to create a new wireless network by emulating a WiFi access point (Soft AP).

wireless→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

wireless→adhocNetwork()YWireless adhocNetwork YWireless

Changes the configuration of the wireless lan interface to create an ad-hoc wireless network, without using an access point.

YWireless **target adhocNetwork ssid securityKey**

On the YoctoHub-Wireless-g, it is best to use softAPNetworkInstead(), which emulates an access point (Soft AP) which is more efficient and more widely supported than ad-hoc networks.

When a security key is specified for an ad-hoc network, the network is protected by a WEP40 key (5 characters or 10 hexadecimal digits) or WEP128 key (13 characters or 26 hexadecimal digits). It is recommended to use a well-randomized WEP128 key using 26 hexadecimal digits to maximize security. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

ssid	the name of the network to connect to
securityKey	the network key, as a character string

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

wireless→get_advertisedValue() YWireless
wireless→advertisedValue()YWireless
get_advertisedValue

Returns the current value of the wireless lan interface (no more than 6 characters).

YWireless **target** **get_advertisedValue**

Returns :
a string corresponding to the current value of the wireless lan interface (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

wireless→get_channel() YWireless
wireless→channel()YWireless get_channel

Returns the 802.11 channel currently used, or 0 when the selected network has not been found.

YWireless **target get_channel**

Returns :

an integer corresponding to the 802.11 channel currently used, or 0 when the selected network has not been found

On failure, throws an exception or returns `Y_CHANNEL_INVALID`.

wireless→get_detectedWlans() YWireless
wireless→detectedWlans()YWireless
get_detectedWlans

Returns a list of YWlanRecord objects that describe detected Wireless networks.

YWireless **target** **get_detectedWlans**

This list is not updated when the module is already connected to an acces 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_linkQuality() YWireless
wireless→linkQuality()YWireless get_linkQuality

Returns the link quality, expressed in percent.

YWireless **target get_linkQuality**

Returns :
an integer corresponding to the link quality, expressed in percent

On failure, throws an exception or returns `Y_LINKQUALITY_INVALID`.

wireless→get_logicalName() YWireless
wireless→logicalName()YWireless get_logicalName

Returns the logical name of the wireless lan interface.

YWireless **target** **get_logicalName**

Returns :
a string corresponding to the logical name of the wireless lan interface.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

wireless→get_message() YWireless
wireless→message()YWireless get_message

Returns the latest status message from the wireless interface.

YWireless **target get_message**

Returns :
a string corresponding to the latest status message from the wireless interface

On failure, throws an exception or returns `Y_MESSAGE_INVALID`.

wireless→get_security() YWireless wireless→security()YWireless get_security

Returns the security algorithm used by the selected wireless network.

YWireless **target** **get_security**

Returns :

a value among `Y_SECURITY_UNKNOWN`, `Y_SECURITY_OPEN`, `Y_SECURITY_WEP`, `Y_SECURITY_WPA` and `Y_SECURITY_WPA2` corresponding to the security algorithm used by the selected wireless network

On failure, throws an exception or returns `Y_SECURITY_INVALID`.

wireless→get_ssid() YWireless
wireless→ssid()YWireless get_ssid

Returns the wireless network name (SSID).

YWireless **target get_ssid**

Returns :

a string corresponding to the wireless network name (SSID)

On failure, throws an exception or returns `Y_SSID_INVALID`.

wireless→joinNetwork()YWireless joinNetwork YWireless

Changes the configuration of the wireless lan interface to connect to an existing access point (infrastructure mode).

YWireless **target joinNetwork ssid securityKey**

Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

ssid	the name of the network to connect to
securityKey	the network key, as a character string

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

wireless→set_logicalName() YWireless
wireless→setLogicalName()YWireless
set_logicalName

Changes the logical name of the wireless lan interface.

YWireless **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 wireless lan interface.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

wireless→softAPNetwork()YWireless softAPNetwork YWireless

Changes the configuration of the wireless lan interface to create a new wireless network by emulating a WiFi access point (Soft AP).

YWireless **target softAPNetwork ssid securityKey**

This function can only be used with the YoctoHub-Wireless-g.

When a security key is specified for a SoftAP network, the network is protected by a WEP40 key (5 characters or 10 hexadecimal digits) or WEP128 key (13 characters or 26 hexadecimal digits). It is recommended to use a well-randomized WEP128 key using 26 hexadecimal digits to maximize security. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

Parameters :

ssid	the name of the network to connect to
securityKey	the network key, as a character string

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

Index

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z