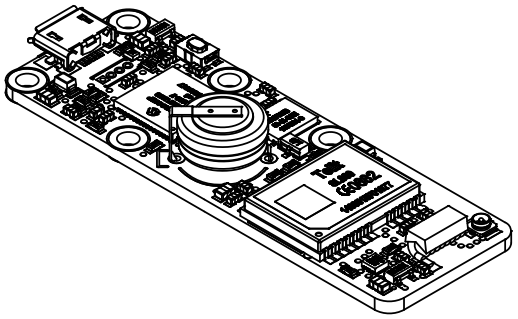
# Yocto-GPS

## User's guide

# Table of contents

# 1. Introduction

The Yocto-GPS is a 60x20mm electronic module equipped with a GNSS (GPS + GLONASS) receiver which enables you to know the position of the receiver, its travel speed, and the current time. It requires the use of an active external antenna with a connector at the U.FL standard.



*The Yocto-GPS module*

The Yocto-GPS is not in itself a complete product. It is a component intended to be integrated into a solution used in laboratory equipments, or in industrial process-control equipments, or for similar applications in domestic and commercial environments. In order to use it, you must at least install it in a protective enclosure and connect it to a host computer.

Yoctopuce thanks you for buying this Yocto-GPS and sincerely hopes that you will be satisfied with it. The Yoctopuce engineers have put a large amount of effort to ensure that your Yocto-GPS is easy to install anywhere and easy to drive from a maximum of programming languages. If you are nevertheless disappointed with this module, or if you need additional information, do not hesitate to contact Yoctopuce support:

| | |
|---|---|
| E-mail address: | support@yoctopuce.com |
| Web site: | www.yoctopuce.com |
| Postal address: | Chemin des Journaliers, 1 |
| ZIP code, city: | 1236 Cartigny |
| Country: | Switzerland |

## 1.1. Safety Information

The Yocto-GPS is designed to meet the requirements of IEC 61010-1:2010 safety standard. It does not create any serious hazards to the operator and surrounding area, even in single fault condition,

as long as it is integrated and used according to the instructions contained in this documentation, and in this section in particular.

### Protective enclosure

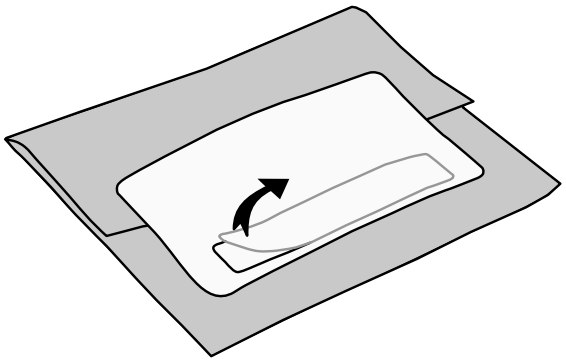
The Yocto-GPS should not be used without a protective enclosure, because of the accessible bare electronic components. For optimal safety, it should be put into a non-metallic, non-inflammable enclosure, resistant to a mechanical stress level of 5 J. For instance, use a polycarbonate (e.g. LEXAN) enclosure rated IK08 with a IEC 60695-11-10 flammability rating of V-1 or better. Using a lower quality enclosure may require specific warnings for the operator and/or compromise conformity with the safety standard.

### Maintenance

If a damage is observed on the electronic board or on the enclosure, it should be replaced in order to ensure continued safety of the equipment, and to prevent damaging other parts of the system due to overload that a short circuit could cause.

### Identification

In order to ease the maintenance and the identification of risks during maintenance, you should affixate the water-resistant identification label provided together with the electronic board as close as possible to the device. If the device is put in a dedicated enclosure, the identification label should be affixated on the outside of the enclosure. This label is resistant to humidity, and can hand rubbing with a piece of cloth soaked with water.



*Identification label is integrated in the package label.*

### Application

The safety standard applied is intended to cover laboratory equipment, industrial process-control equipment and similar applications in residential or commercial environment. If you intend to use the Yocto-GPS for another kind of application, you should check the safety regulations according to the standard applicable to your application.

In particular, the Yocto-GPS is *not* certified for use in medical environments or for life-support applications.

### Environment

The Yocto-GPS is *not* certified for use in hazardous locations, explosive environments, or life-threatening applications. Environmental ratings are provided below.

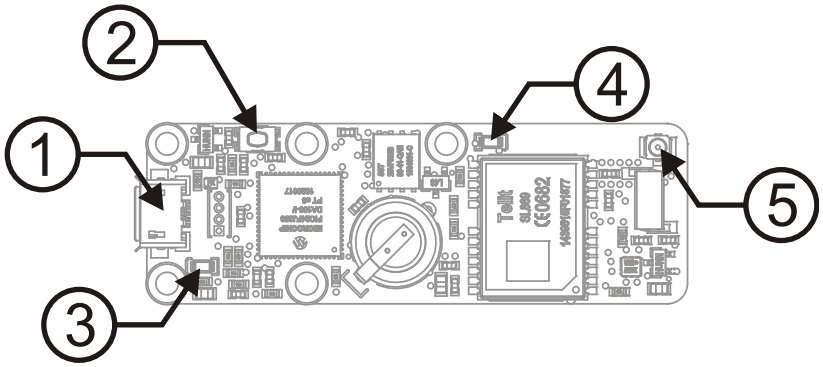
## 1.2. Environmental conditions

Yoctopuce devices have been designed for indoor use in a standard office or laboratory environment (IEC 60664 *pollution degree 2*): air pollution is expected to be limited and mainly non-conductive. Relative humidity is expected to be between 10% and 90% RH, non condensing. Use in environments with significant solid pollution or conductive pollution requires a protection from such pollution using an IP67 or IP68 enclosure. The products are designed for use up to altitude 2000m.

All Yoctopuce devices are warranted to perform according to their documentation and technical specifications under normal temperature conditions according to IEC61010-1, i.e. 5°C to 40°C. In addition, most devices can also be used on an extended temperature range, where some limitations may apply from case to case.

The extended operating temperature range for the Yocto-GPS is -25...85°C. This temperature range has been determined based on components manufacturer recommendations, and on controlled environment tests performed during a limited duration (1h). If you plan to use the Yocto-GPS in harsh environments for a long period of time, we strongly advise you to run extensive tests before going to production.

# 2. Presentation




*1: Micro-B USB socket*
*2: Yocto-button*
*3: Yocto-led*
*4: Satellite reception led*
*5: Antenna connector*

## 2.1. Common elements

All Yocto-modules share a number of common functionalities.

### USB connector

Yoctopuce modules all come with a USB 2.0 micro-B socket. Warning: the USB connector is simply soldered in surface and can be pulled out if the USB plug acts as a lever. In this case, if the tracks stayed in position, the connector can be soldered back with a good iron and using flux to avoid bridges. Alternatively, you can solder a USB cable directly in the 1.27mm-spaced holes near the connector.

If you plan to use a power source other then a standard USB host port to power the device through the USB connector, that power source must respect the assigned values of USB 2.0 specifications:

- **Voltage min.:** 4.75 V DC
- **Voltage max.:** 5.25 V DC
- **Over-current protection:** 5.0 A max.

A higher voltage is likely to destroy the device. THe behaviour with a lower voltage is not specified, but it can result firmware corruption.

### Yocto-button

The Yocto-button has two functionalities. First, it can activate the Yocto-beacon mode (see below under Yocto-led). Second, if you plug in a Yocto-module while keeping this button pressed, you can then reprogram its firmware with a new version. Note that there is a simpler UI-based method to update the firmware, but this one works even in case of severely damaged firmware.

### Yocto-led

Normally, the Yocto-led is used to indicate that the module is working smoothly. The Yocto-led then emits a low blue light which varies slowly, mimicking breathing. The Yocto-led stops breathing when the module is not communicating any more, as for instance when powered by a USB hub which is disconnected from any active computer.

When you press the Yocto-button, the Yocto-led switches to Yocto-beacon mode. It starts flashing faster with a stronger light, in order to facilitate the localization of a module when you have several identical ones. It is indeed possible to trigger off the Yocto-beacon by software, as it is possible to detect by software that a Yocto-beacon is on.

The Yocto-led has a third functionality, which is less pleasant: when the internal software which controls the module encounters a fatal error, the Yocto-led starts emitting an SOS in morse [1]. If this happens, unplug and re-plug the module. If it happens again, check that the module contains the latest version of the firmware, and, if it is the case, contact Yoctopuce support[2].

### Current sensor

Each Yocto-module is able to measure its own current consumption on the USB bus. Current supply on a USB bus being quite critical, this functionality can be of great help. You can only view the current consumption of a module by software.

### Serial number

Each Yocto-module has a unique serial number assigned to it at the factory. For Yocto-GPS modules, this number starts with YGNSSMK1. The module can be software driven using this serial number. The serial number cannot be modified.

### Logical name

The logical name is similar to the serial number: it is a supposedly unique character string which allows you to reference your module by software. However, in the opposite of the serial number, the logical name can be modified at will. The benefit is to enable you to build several copies of the same project without needing to modify the driving software. You only need to program the same logical name in each copy. Warning: the behavior of a project becomes unpredictable when it contains several modules with the same logical name and when the driving software tries to access one of these modules through its logical name. When leaving the factory, modules do not have an assigned logical name. It is yours to define.

## 2.2. Specific elements

### The receiver

The Yocto-GPS is based on a GNSS SL869 receiver produced by Telit. It has 32 channels and can compute up to 10 positions per second.

### Satellite reception led

The green led (Fix Sat.) on the module indicates the state of the satellite reception. It stays on when the reception is satisfactory. It blinks fast when the module searches for satellites. With an appropriate reception, the Yocto-GPS can obtain a position in less than 35 seconds after a cold start. Moreover, the Yocto-GPS has a small battery enabling the GPS to remember during several hours

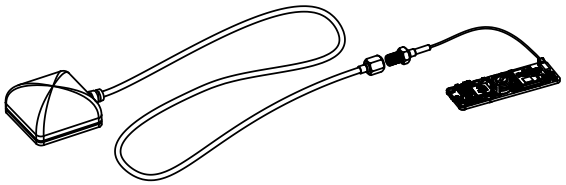
[1] short-short-short long-long-long short-short-short
[2] support@yoctopuce.com

the position of the last seen satellites, enabling a hot start in a few seconds only in most cases. In case of a particularly degraded satellite reception, obtaining a first position can take up to 5 minutes.

### The antenna connector

The Yocto-GPS needs an antenna to work. To maximize reception quality, the Yocto-GPS is designed to work with an active external antenna (powered seamlessly in 3.3V), connected with a U.FL connector. Beware, U.FL connectors are not designed to be connected/disconnected continuously. If you plan such a scenario, use a small U.FL to SMA (female) cable and an antenna equipped with an SMA connector.

Being active, you can deport the antenna several meters away without changing the quality of the reception signal.



*Use an SMA connector if you plan to frequently disconnect the antenna.*

## 2.3. Optional accessories

The accessories below are not necessary to use the Yocto-GPS module but might be useful depending on your project. These are mostly common products that you can buy from your favorite hacking store. To save you the tedious job of looking for them, most of them are also available on the Yoctopuce shop.

### Screws and spacers

In order to mount the Yocto-GPS module, you can put small screws in the 2.5mm assembly holes, with a screw head no larger than 4.5mm. The best way is to use threaded spacers, which you can then mount wherever you want. You can find more details on this topic in the chapter about assembly and connections.

### Micro-USB hub

If you intend to put several Yoctopuce modules in a very small space, you can connect them directly to a micro-USB hub. Yoctopuce builds a USB hub particularly small for this purpose (down to 20mmx36mm), on which you can directly solder a USB cable instead of using a USB plug. For more details, see the micro-USB hub information sheet.

### YoctoHub-Ethernet, YoctoHub-Wireless and YoctoHub-GSM

You can add network connectivity to your Yocto-GPS, thanks to the YoctoHub-Ethernet, the YoctoHub-Wireless and the YoctoHub-GSM which provides repectiveley Ethernet, WiFi and GSM connectivity. All of them can drive up to three devices and behave exactly like a regular computer running a *VirtualHub*.

### 1.27mm (or 1.25mm) connectors

In case you wish to connect your Yocto-GPS to a Micro-hub USB or a YoctoHub without using a bulky USB connector, you can use the four 1.27mm pads just behind the USB connector. There are two options.

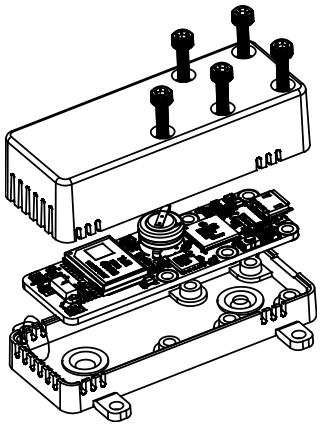
You can mount the Yocto-GPS directly on the hub using screw and spacers, and connect it using 1.27mm board-to-board connectors. To prevent shortcuts, it is best to solder the female connector on the hub and the male connector on the Yocto-GPS.

You can also use a small 4-wires cable with a 1.27mm connector. 1.25mm works as well, it does not make a difference for 4 pins. This makes it possible to move the device a few inches away. Don't put

it too far away if you use that type of cable, because as the cable is not shielded, it may cause undesirable electromagnetic emissions.

## Enclosure

Your Yocto-GPS has been designed to be installed as is in your project. Nevertheless, Yoctopuce sells enclosures specifically designed for Yoctopuce devices. These enclosures have removable mounting brackets and magnets allowing them to stick on ferromagnetic surfaces. More details are available on the Yoctopuce web site [3]. The suggested enclosure model for your Yocto-GPS is the YoctoBox-Long-Thick-Black.



*You can install your Yocto-GPS in an optional enclosure*

[3] http://www.yoctopuce.com/EN/products/category/enclosures

# 3. First steps

By design, all Yoctopuce modules are driven the same way. Therefore, user's guides for all the modules of the range are very similar. If you have already carefully read through the user's guide of another Yoctopuce module, you can jump directly to the description of the module functions.

## 3.1. Prerequisites

In order to use your Yocto-GPS module, you should have the following items at hand.

### A computer

Yoctopuce modules are intended to be driven by a computer (or possibly an embedded microprocessor). You will write the control software yourself, according to your needs, using the information provided in this manual.

Yoctopuce provides software libraries to drive its modules for the following operating systems: Windows, macOS X, Linux, and Android. Yoctopuce modules do not require installing any specific system driver, as they leverage the standard HID driver[1] provided with every operating system.

Windows versions currently supported are: Windows XP, Windows 2003, Windows Vista, Windows 7, Windows 8 and Windows 10. Both 32 bit and 64 bit versions are supported. The programming library is also available for the Universal Windows Platform (UWP), which is supported by all flavors of Windows 10, including Windows 10 IoT. Yoctopuce is frequently testing its modules on Windows 7 and Windows 10.

MacOS versions currently supported are: Mac OS X 10.9 (Maverick), 10.10 (Yosemite), 10.11 (El Capitan), macOS 10.12 (Sierra), macOS 10.13 (High Sierra) and macOS 10.14 (Mojave). Yoctopuce is frequently testing its modules on macOS 10.14.

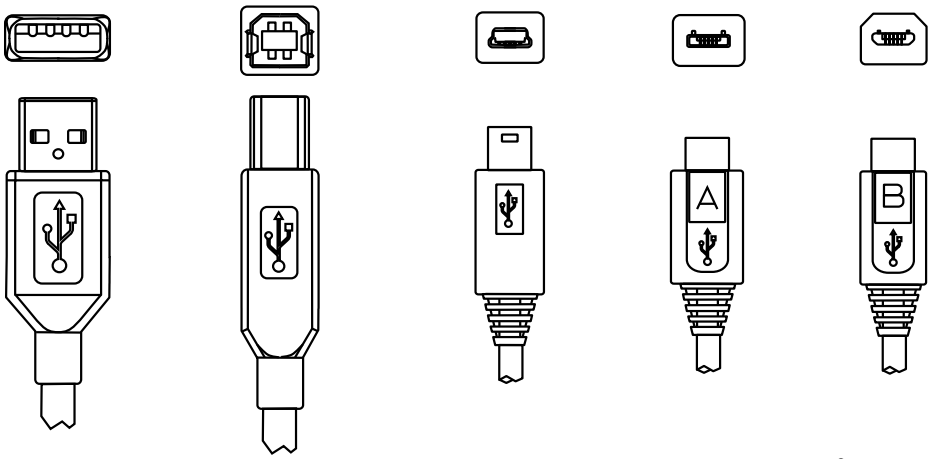
Linux kernels currently supported are the 2.6 branch, the 3.x branch and the 4.x branch. Other versions of the Linux kernel, and even other UNIX variants, are very likely to work as well, as Linux support is implemented through the standard **libusb** API. Yoctopuce is frequently testing its modules on Linux kernel 4.15 (Ubuntu 18.04 LTS).

Android versions currently supported are: Android 3.1 and later. Moreover, it is necessary for the tablet or phone to support the *Host* USB mode. Yoctopuce is frequently testing its modules on Android 7.x on a Samsung Galaxy A6 with the Java for Android library.

[1] The HID driver is the one that takes care of the mouse, the keyboard, etc.

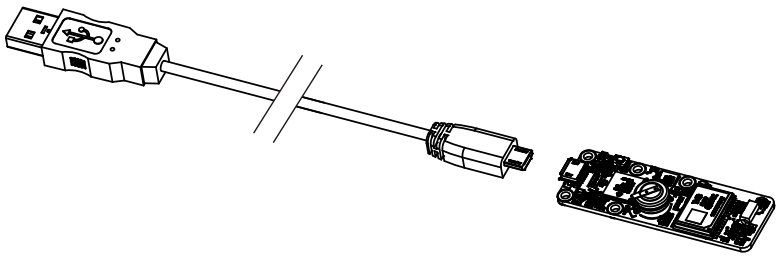
### A USB 2.0 cable, type A-micro B

USB 2.0 connectors exist in three sizes: the "standard" size that you probably use to connect your printer, the very common mini size to connect small devices, and finally the micro size often used to connect mobile phones, as long as they do not exhibit an apple logo. All USB modules manufactured by Yoctopuce use micro size connectors.




*The most common USB 2.0 connectors: A, B, Mini B, Micro A, Micro B*[2]

To connect your Yocto-GPS module to a computer, you need a USB 2.0 cable of type A-micro B. The price of this cable may vary a lot depending on the source, look for it under the name *USB 2.0 A to micro B Data cable*. Make sure not to buy a simple USB charging cable without data connectivity. The correct type of cable is available on the Yoctopuce shop.



*You must plug in your Yocto-GPS module with a USB 2.0 cable of type A - micro B*

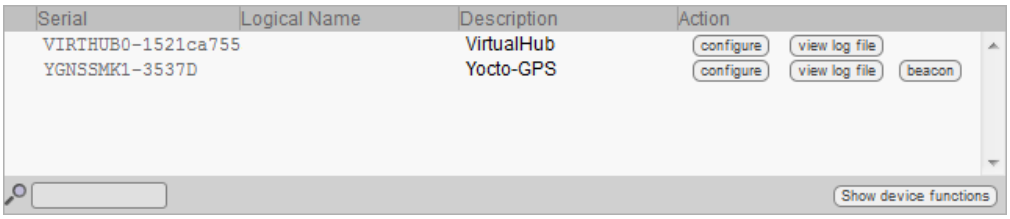
If you insert a USB hub between the computer and the Yocto-GPS module, make sure to take into account the USB current limits. If you do not, be prepared to face unstable behaviors and unpredictable failures. You can find more details on this topic in the chapter about assembly and connections.

## 3.2. Testing USB connectivity

At this point, your Yocto-GPS should be connected to your computer, which should have recognized it. It is time to make it work.

Go to the Yoctopuce web site and download the *Virtual Hub* software[3]. It is available for Windows, Linux, and Mac OS X. Normally, the Virtual Hub software serves as an abstraction layer for languages which cannot access the hardware layers of your computer. However, it also offers a succinct interface to configure your modules and to test their basic functions. You access this interface with a simple web browser[4]. Start the *Virtual Hub* software in a command line, open your preferred web browser and enter the URL *http://127.0.0.1:4444*. The list of the Yoctopuce modules connected to your computer is displayed.

[2] *Although they existed for some time, Mini A connectors are not available anymore http://www.usb.org/developers/Deprecation_Announcement_052507.pdf*
[3] www.yoctopuce.com/EN/virtualhub.php
[4] The interface is tested on Chrome, FireFox, Safari, Edge et IE 11.



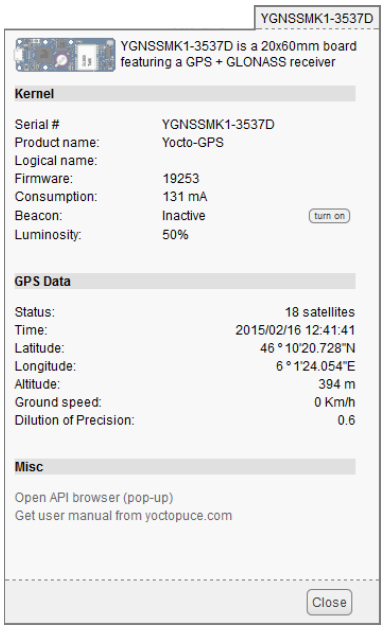

*Module list as displayed in your web bowser*

## 3.3. Localization

You can then physically localize each of the displayed modules by clicking on the **beacon** button. This puts the Yocto-led of the corresponding module in Yocto-beacon mode. It starts flashing, which allows you to easily localize it. The second effect is to display a little blue circle on the screen. You obtain the same behavior when pressing the Yocto-button of the module.

## 3.4. Test of the module

The first item to check is that your module is working well: click on the serial number corresponding to your module. This displays a window summarizing the properties of your Yocto-GPS.

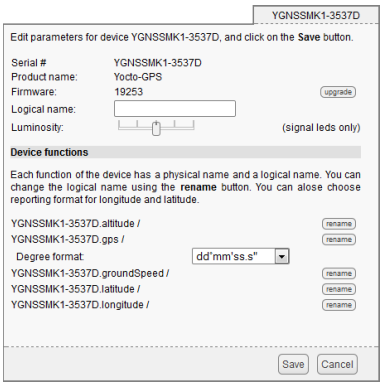



*Properties of the Yocto-GPS module*

This window allows you, among other things, to play with you module to check how it is working. Position values computed by the Yocto-GPS are displayed there in real time.

## 3.5. Configuration

When, in the module list, you click on the **configure** button corresponding to your module, the configuration window is displayed.




*Yocto-GPS module configuration.*

## Firmware

The module firmware can easily be updated with the help of the interface. Firmware destined for Yoctopuce modules are available as .byn files and can be downloaded from the Yoctopuce web site.

To update a firmware, simply click on the **upgrade** button on the configuration window and follow the instructions. If the update fails for one reason or another, unplug and re-plug the module and start the update process again. This solves the issue in most cases. If the module was unplugged while it was being reprogrammed, it does probably not work anymore and is not listed in the interface. However, it is always possible to reprogram the module correctly by using the *Virtual Hub* software [5] in command line [6].

## Logical name of the module

The logical name is a name that you choose, which allows you to access your module, in the same way a file name allows you to access its content. A logical name has a maximum length of 19 characters. Authorized characters are `A..Z`, `a..z`, `0..9`, `_`, and `-`. If you assign the same logical name to two modules connected to the same computer and you try to access one of them through this logical name, behavior is undetermined: you have no way of knowing which of the two modules answers.

## Luminosity

This parameter allows you to act on the maximal intensity of the leds of the module. This enables you, if necessary, to make it a little more discreet, while limiting its power consumption. Note that this parameter acts on all the signposting leds of the module, including the Yocto-led. If you connect a module and no led turns on, it may mean that its luminosity was set to zero.

## Logical names of functions

Each Yoctopuce module has a serial number and a logical name. In the same way, each function on each Yoctopuce module has a hardware name and a logical name, the latter can be freely chosen by the user. Using logical names for functions provides a greater flexibility when programming modules.

You can assign logical names to several functions of the Yocto-GPS by clicking on the corresponding "rename" button.

## The GPS function

The **gps** function groups all the measures computed by the module: position, ground speed, travel direction, time, and so on... Latitude and longitude are presented as character strings with a format to be selected between DDï¿½MM'SS.SSS", DDï¿½MM.MMMM, and DD.DDDDDD. You can make do with only this function. But if you need to make calculations on the positions, or if you want to use callbacks[7], the following functions prove useful.

[5] www.yoctopuce.com/EN/virtualhub.php
[6] More information available in the virtual hub documentation
[7] See the chapter on "Advanced programming"

## Latitude and Longitude functions

They enable you to obtain the current latitude and longitude in numerical format, independently of the display format selected in the GPS function. Note, for technical reasons, the values are not given in degrees but in millidegrees (thousandths of degree).

## The Altitude function

It returns an estimate of the current altitude. Beware, altitude computations based on GPS systems are very imprecise. Expect errors of several tens of meters.

## The groundSpeed function

The Yocto-GPS can compute the ground speed, that is the horizontal speed relative to the ground.

## Limitations

The Yocto-GPS has pretty standard limitations: it will lose track if any of the following limits are exceeded:

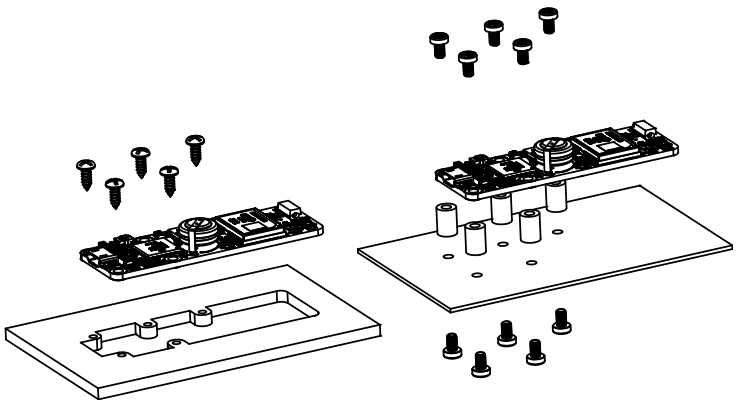
- ITAR limits: velocity greater than 515 m/s AND altitude above 18,000 m
- altitude: 100,000 m (max) or -1500 m (min)
- velocity: 600 m/s (max)
- acceleration: 2g (max)

# 4. Assembly and connections

This chapter provides important information regarding the use of the Yocto-GPS module in real-world situations. Make sure to read it carefully before going too far into your project if you want to avoid pitfalls.

## 4.1. Fixing

While developing your project, you can simply let the module hang at the end of its cable. Check only that it does not come in contact with any conducting material (such as your tools). When your project is almost at an end, you need to find a way for your modules to stop moving around.



*Examples of assembly on supports*

The Yocto-GPS module contains 2.5mm assembly holes. You can use these holes for screws. The screw head diameter must not be larger than 4.5mm or they will damage the module circuits. Make sure that the lower surface of the module is not in contact with the support. We recommend using spacers, but other methods are possible. Nothing prevents you from fixing the module with a glue gun; it will not be good-looking, but it will hold.

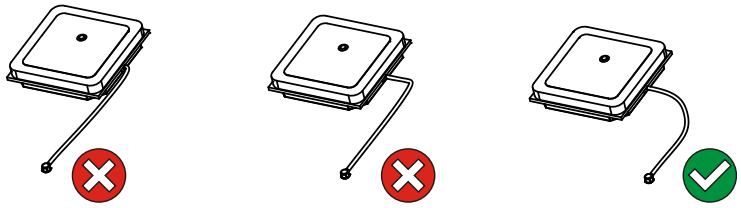
If your intend to screw your module directly against a conducting part, for example a metallic frame, insert an isolating layer in between. Otherwise you are bound to induce a short circuit: there are naked pads under your module. Simple insulating tape should be enough.

## 4.2. Antenna

The antenna is a key element of your system. Make sure that it is always set up in such a way as it can "see" the largest part of the sky possible. Except in particular cases, you cannot receive good quality GPS signals from inside. However, if you really want to try, set the antenna near a window.

### Antenna cable

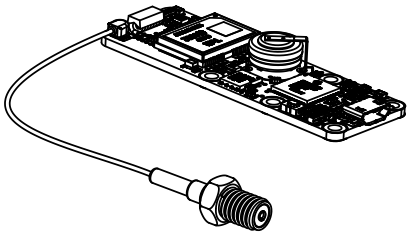
The Yocto-GPS is sold with a ceramic active antenna at the end of a ten centimeter coaxial cable ending with a U.FL connector. This type of cable is rather fragile: avoid bending it at a right angle, you could damage it. Be particularly careful about the antenna-cable juncture.



*Coaxial antenna cables are fragile: avoid bending them*

### U.FL connectors

Theoretically, U.FL connectors are not designed to be connected more than about 10 times. Make sure that the U.FL plug offers a good contact: when connected, it must not be able to rotate freely around the connector. If you plan to build a system with a removable external antenna, use a female U.FL to SMA adaptor cable and a corresponding antenna. Almost all external antennas have an SMA connector.



*External GPS antenna usually have an SMA connector*

### Choosing an antenna

You do not need to use the GPS antenna delivered with the Yocto-GPS. There are many GPS antennas on the market. You must select an active antenna: active antennas include a small electronic amplifier limiting the loses due to the antenna cable. They are usually required when the distance between the antenna and the GPS receiver goes above 10 centimeters or so. An active GPS antenna is most often powered directly through the antenna cable. You can use any active GPS antenna directly with the Yocto-GPS as long as

- it can work when powered in 3.3V
- its gain does not go above 35db.

## 4.3. USB power distribution

Although USB means *Universal Serial BUS*, USB devices are not physically organized as a flat bus but as a tree, using point-to-point connections. This has consequences on power distribution: to make it simple, every USB port must supply power to all devices directly or indirectly connected to it. And USB puts some limits.

In theory, a USB port provides 100mA, and may provide up to 500mA if available and requested by the device. In the case of a hub without external power supply, 100mA are available for the hub itself, and the hub should distribute no more than 100mA to each of its ports. This is it, and this is not much. In particular, it means that in theory, it is not possible to connect USB devices through two cascaded hubs without external power supply. In order to cascade hubs, it is necessary to use self-powered USB hubs, that provide a full 500mA to each subport.

In practice, USB would not have been as successful if it was really so picky about power distribution. As it happens, most USB hub manufacturers have been doing savings by not implementing current limitation on ports: they simply connect the computer power supply to every port, and declare themselves as *self-powered hub* even when they are taking all their power from the USB bus (in

order to prevent any power consumption check in the operating system). This looks a bit dirty, but given the fact that computer USB ports are usually well protected by a hardware current limitation around 2000mA, it actually works in every day life, and seldom makes hardware damage.

What you should remember: if you connect Yoctopuce modules through one, or more, USB hub without external power supply, you have no safe-guard and you depend entirely on your computer manufacturer attention to provide as much current as possible on the USB ports, and to detect overloads before they lead to problems or to hardware damages. When modules are not provided enough current, they may work erratically and create unpredictable bugs. If you want to prevent any risk, do not cascade hubs without external power supply, and do not connect peripherals requiring more than 100mA behind a bus-powered hub.

In order to help you controlling and planning overall power consumption for your project, all Yoctopuce modules include a built-in current sensor that indicates (with 5mA precision) the consumption of the module on the USB bus.

Note also that the USB cable itself may also cause power supply issues, in particular when the wires are too thin or when the cable is too long [1]. Good cables are usually made using AWG 26 or AWG 28 wires for data lines and AWG 24 wires for power.

## 4.4. Electromagnetic compatibility (EMI)

Connection methods to integrate the Yocto-GPS obviously have an impact on the system overall electromagnetic emissions, and therefore also impact the conformity with international standards.

When we perform reference measurements to validate the conformity of our products with IEC CISPR 11, we do not use any enclosure but connect the devices using a shielded USB cable, compliant with USB 2.0 specifications: the cable shield is connected to both connector shells, and the total resistance from shell to shell is under 0.6Ω. The USB cable length is 3m, in order to expose one meter horizontally, one meter vertically and keep the last meter close to the host computer within a ferrite bead.

If you use a non-shielded USB cable, or an improperly shielded cable, your system will work perfectly well but you may not remain in conformity with the emission standard. If you are building a system made of multiple devices connected using 1.27mm pitch connectors, or with a sensor moved away from the device CPU, you can generally recover the conformity by using a metallic enclosure acting as an external shield.

Still on the topic of electromagnetic compatibility, the maximum supported length of the USB cable is 3m. In addition to the voltage drop issue mentionned above, using longer wires would require to run extra tests to assert compatibility with the electromagnetic immunity standards.

[1] www.yoctopuce.com/EN/article/usb-cables-size-matters

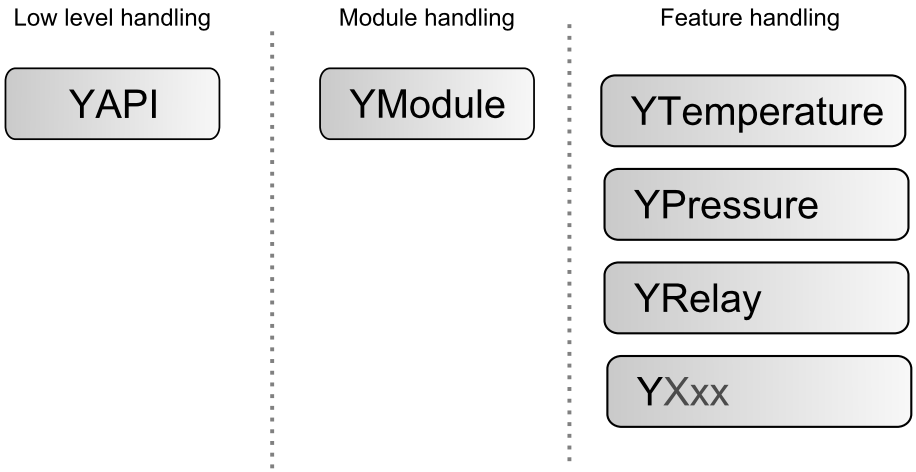
# 5. Programming, general concepts

The Yoctopuce API was designed to be at the same time simple to use and sufficiently generic for the concepts used to be valid for all the modules in the Yoctopuce range, and this in all the available programming languages. Therefore, when you have understood how to drive your Yocto-GPS with your favorite programming language, learning to use another module, even with a different language, will most likely take you only a minimum of time.

## 5.1. Programming paradigm

The Yoctopuce API is object oriented. However, for simplicity's sake, only the basics of object programming were used. Even if you are not familiar with object programming, it is unlikely that this will be a hinderance for using Yoctopuce products. Note that you will never need to allocate or deallocate an object linked to the Yoctopuce API: it is automatically managed.

There is one class per Yoctopuce function type. The name of these classes always starts with a Y followed by the name of the function, for example *YTemperature*, *YRelay*, *YPressure*, etc.. There is also a *YModule* class, dedicated to managing the modules themselves, and finally there is the static YAPI class, that supervises the global workings of the API and manages low level communications.




*Structure of the Yoctopuce API.*

### The YSensor class

Each Yoctopuce sensor function has its dedicated class: YTemperature to measure the temperature, YVoltage to measure a voltage, YRelay to drive a relay, etc. However there is a special class that can do more: YSensor.

The YSensor class is the parent class for all Yoctopuce sensors, and can provide access to any sensor, regardless of its type. It includes methods to access all common functions. This makes it easier to create applications that use many different sensors. Moreover, if you create an application based on YSensor, it will work with all Yoctopuce sensors, even those which do no yet exist.

## Programmation

In the Yoctopuce API, priority was put on the ease of access to the module functions by offering the possibility to make abstractions of the modules implementing them. Therefore, it is quite possible to work with a set of functions without ever knowing exactly which module are hosting them at the hardware level. This tremendously simplifies programming projects with a large number of modules.

From the programming stand point, your Yocto-GPS is viewed as a module hosting a given number of functions. In the API, these functions are objects which can be found independently, in several ways.

## Access to the functions of a module

**Access by logical name**
Each function can be assigned an arbitrary and persistent logical name: this logical name is stored in the flash memory of the module, even if this module is disconnected. An object corresponding to an *Xxx* function to which a logical name has been assigned can then be directly found with this logical name and the *YXxx.FindXxx* method. Note however that a logical name must be unique among all the connected modules.

**Access by enumeration**
You can enumerate all the functions of the same type on all the connected modules with the help of the classic enumeration functions *FirstXxx* and *nextXxxx* available for each *YXxx* class.

**Access by hardware name**
Each module function has a hardware name, assigned at the factory and which cannot be modified. The functions of a module can also be found directly with this hardware name and the *YXxx.FindXxx* function of the corresponding class.

**Difference between *Find* and *First***
The *YXxx.FindXxxx* and *YXxx.FirstXxxx* methods do not work exactly the same way. If there is no available module, *YXxx.FirstXxxx* returns a null value. On the opposite, even if there is no corresponding module, *YXxx.FindXxxx* returns a valid object, which is not online but which could become so if the corresponding module is later connected.

## Function handling

When the object corresponding to a function is found, its methods are available in a classic way. Note that most of these subfunctions require the module hosting the function to be connected in order to be handled. This is generally not guaranteed, as a USB module can be disconnected after the control software has started. The *isOnline* method, available in all the classes, is then very helpful.

## Access to the modules

Even if it is perfectly possible to build a complete project while making a total abstraction of which function is hosted on which module, the modules themselves are also accessible from the API. In fact, they can be handled in a way quite similar to the functions. They are assigned a serial number at the factory which allows you to find the corresponding object with *YModule.Find()*. You can also assign arbitrary logical names to the modules to make finding them easier. Finally, the *YModule* class contains the *YModule.FirstModule()* and *nextModule()* enumeration methods allowing you to list the connected modules.

### Functions/Module interaction

From the API standpoint, the modules and their functions are strongly uncorrelated by design. Nevertheless, the API provides the possibility to go from one to the other. Thus, the *get_module()* method, available for each function class, allows you to find the object corresponding to the module hosting this function. Inversely, the *YModule* class provides several methods allowing you to enumerate the functions available on a module.

## 5.2. The Yocto-GPS module

The Yocto-GPS is an GNSS interface .

**module : Module**

| attribute | type | modifiable ? |
|---|---|---|
| productName | String | read-only |
| serialNumber | String | read-only |
| logicalName | String | read-only |
| productId | Hexadecimal number | read-only |
| productRelease | Hexadecimal number | read-only |
| firmwareRelease | String | read-only |
| persistentSettings | Enumerated | read-only |
| luminosity | 0..100% | read-only |
| beacon | On/Off | read-only |
| upTime | Time | read-only |
| usbCurrent | Used current (mA) | read-only |
| rebootCountdown | Integer | read-only |
| userVar | Integer | read-only |

**gps : Gps**

| attribute | type | modifiable ? |
|---|---|---|
| logicalName | String | read-only |
| advertisedValue | String | read-only |
| isFixed | Boolean | read-only |
| satCount | Integer | read-only |
| satPerConst | Integer | read-only |
| gpsRefreshRate | Fixed-point number | read-only |
| coordSystem | Enumerated | read-only |
| constellation | Enumerated | read-only |
| latitude | String | read-only |
| longitude | String | read-only |
| dilution | Fixed-point number | read-only |
| altitude | Fixed-point number | read-only |
| groundSpeed | Fixed-point number | read-only |
| direction | Fixed-point number | read-only |
| unixTime | UTC time | read-only |
| dateTime | String | read-only |
| utcOffset | Integer | read-only |
| command | String | read-only |

**latitude : Latitude**

| attribute | type | modifiable ? |
|---|---|---|
| logicalName | String | read-only |
| advertisedValue | String | read-only |
| unit | String | read-only |
| currentValue | Fixed-point number | read-only |
| lowestValue | Fixed-point number | read-only |
| highestValue | Fixed-point number | read-only |
| currentRawValue | Fixed-point number | read-only |
| logFrequency | Frequency | read-only |

| | | |
|---|---|---|
| reportFrequency | Frequency | read-only |
| advMode | Enumerated | read-only |
| calibrationParam | Calibration parameters | read-only |
| resolution | Fixed-point number | read-only |
| sensorState | Integer | read-only |

**longitude : Longitude**

| attribute | type | modifiable ? |
|---|---|---|
| logicalName | String | read-only |
| advertisedValue | String | read-only |
| unit | String | read-only |
| currentValue | Fixed-point number | read-only |
| lowestValue | Fixed-point number | read-only |
| highestValue | Fixed-point number | read-only |
| currentRawValue | Fixed-point number | read-only |
| logFrequency | Frequency | read-only |
| reportFrequency | Frequency | read-only |
| advMode | Enumerated | read-only |
| calibrationParam | Calibration parameters | read-only |
| resolution | Fixed-point number | read-only |
| sensorState | Integer | read-only |

**altitude : Altitude**

| attribute | type | modifiable ? |
|---|---|---|
| logicalName | String | read-only |
| advertisedValue | String | read-only |
| unit | String | read-only |
| currentValue | Fixed-point number | read-only |
| lowestValue | Fixed-point number | read-only |
| highestValue | Fixed-point number | read-only |
| currentRawValue | Fixed-point number | read-only |
| logFrequency | Frequency | read-only |
| reportFrequency | Frequency | read-only |
| advMode | Enumerated | read-only |
| calibrationParam | Calibration parameters | read-only |
| resolution | Fixed-point number | read-only |
| sensorState | Integer | read-only |
| qnh | Fixed-point number | read-only |
| technology | String | read-only |

**groundSpeed : GroundSpeed**

| attribute | type | modifiable ? |
|---|---|---|
| logicalName | String | read-only |
| advertisedValue | String | read-only |
| unit | String | read-only |
| currentValue | Fixed-point number | read-only |
| lowestValue | Fixed-point number | read-only |
| highestValue | Fixed-point number | read-only |
| currentRawValue | Fixed-point number | read-only |
| logFrequency | Frequency | read-only |
| reportFrequency | Frequency | read-only |
| advMode | Enumerated | read-only |
| calibrationParam | Calibration parameters | read-only |
| resolution | Fixed-point number | read-only |
| sensorState | Integer | read-only |

**dataLogger : DataLogger**

| attribute | type | modifiable ? |
|---|---|---|

| | | |
|---|---|---|
| logicalName | String | read-only |
| advertisedValue | String | read-only |
| currentRunIndex | Integer | read-only |
| timeUTC | UTC time | read-only |
| recording | Enumerated | read-only |
| autoStart | On/Off | read-only |
| beaconDriven | On/Off | read-only |
| usage | 0..100% | read-only |
| clearHistory | Boolean | read-only |

## 5.3. Module

Global parameters control interface for all Yoctopuce devices

The YModule class can be used with all Yoctopuce USB devices. It can be used to control the module global parameters, and to enumerate the functions provided by each module.

### productName

Character string containing the commercial name of the module, as set by the factory.

### serialNumber

Character string containing the serial number, unique and programmed at the factory. For a Yocto-GPS module, this serial number always starts with YGNSSMK1. You can use the serial number to access a given module by software.

### logicalName

Character string containing the logical name of the module, initially empty. This attribute can be modified at will by the user. Once initialized to an non-empty value, it can be used to access a given module. If two modules with the same logical name are in the same project, there is no way to determine which one answers when one tries accessing by logical name. The logical name is limited to 19 characters among A..Z,a..z,0..9,_, and -.

### productId

USB device identifier of the module, preprogrammed to 83 at the factory.

### productRelease

Release number of the module hardware, preprogrammed at the factory. The original hardware release returns value 1, revision B returns value 2, etc.

### firmwareRelease

Release version of the embedded firmware, changes each time the embedded software is updated.

### persistentSettings

State of persistent module settings: loaded from flash memory, modified by the user or saved to flash memory.

### luminosity

Lighting strength of the informative leds (e.g. the Yocto-Led) contained in the module. It is an integer value which varies between 0 (LEDs turned off) and 100 (maximum led intensity). The default value is 50. To change the strength of the module LEDs, or to turn them off completely, you only need to change this value.

### beacon

Activity of the localization beacon of the module.

### upTime

Time elapsed since the last time the module was powered on.

### usbCurrent

Current consumed by the module on the USB bus, in milli-amps.

### rebootCountdown

Countdown to use for triggering a reboot of the module.

### userVar

32bit integer variable available for user storage.

## 5.4. Gps

Geolocalization control interface (GPS, GNSS, ...), available for instance in the Yocto-GPS

The `YGps` class allows you to retrieve positioning data from a GPS/GNSS sensor. This class can provides complete positioning information. However, if you wish to define callbacks on position changes or record the position in the datalogger, you should use the `YLatitude` et `YLongitude` classes.

### logicalName

Character string containing the logical name of the geolocalization module, initially empty. This attribute can be modified at will by the user. Once initialized to an non-empty value, it can be used to access the geolocalization module directly. If two geolocalization modules with the same logical name are used in the same project, there is no way to determine which one answers when one tries accessing by logical name. The logical name is limited to 19 characters among `A..Z`,`a..z`,`0..9`,`_`, and `-`.

### advertisedValue

Short character string summarizing the current GPS reception state.

### isFixed

Indicates whether the receiver has found enough satellites to work

### satCount

Satellite count used to compute GPS position .

### satPerConst

Visible satellite count per constellation

### gpsRefreshRate

GPS receiver current refresh rate.

### coordSystem

Representation system to use for positioning data

### constellation

Satellite constellations to use for positioning computation

### latitude

Current latitude

### longitude

Current longitude

### dilution

Current dilution of precision

### altitude

Current altitude

### groundSpeed

Current ground speed

### direction

Current direction

### unixTime

Unix form of the current time (number of elapsed seconds since Jan 1970 1st)

### dateTime

Current time in the form "YYYY/MM/DD hh:mm:ss"

### utcOffset

Number of seconds between current time and UTC time (time zone)

### command

Magic attribute used to send commands to the device. If a command is not interpreted as expected, check the device logs.

## 5.5. Latitude

latitude sensor control interface, available for instance in the Yocto-GPS

The `YLatitude` class allows you to read and configure Yoctopuce latitude sensors. It inherits from `YSensor` class the core functions to read measurements, to register callback functions, and to access the autonomous datalogger.

### logicalName

Character string containing the logical name of the latitude sensor, initially empty. This attribute can be modified at will by the user. Once initialized to an non-empty value, it can be used to access the latitude sensor directly. If two latitude sensors with the same logical name are used in the same project, there is no way to determine which one answers when one tries accessing by logical name. The logical name is limited to 19 characters among `A..Z`,`a..z`,`0..9`,`_`, and `-`.

### advertisedValue

Short character string summarizing the current state of the latitude sensor, that is automatically advertised up to the parent hub. For a latitude sensor, the advertised value is the current value of the latitude.

### unit

Short character string representing the measuring unit for the latitude.

### currentValue

Current value of the latitude, in deg/1000, as a floating point number.

### lowestValue

Minimal value of the latitude, in deg/1000, as a floating point number.

### highestValue

Maximal value of the latitude, in deg/1000, as a floating point number.

### currentRawValue

Uncalibrated, unrounded raw value returned by the sensor, as a floating point number.

### logFrequency

Datalogger recording frequency, or "OFF" when measures should not be stored in the data logger flash memory.

### reportFrequency

Timed value notification frequency, or "OFF" when timed value notifications are disabled for this function.

### advMode

Measuring mode for the advertised value pushed to the parent hub.

### calibrationParam

Extra calibration parameters (for instance to compensate for the effects of an enclosure), as an array of 16 bit words.

### resolution

Measure resolution (i.e. precision of the numeric representation, not necessarily of the measure itself).

### sensorState

Sensor health state (zero when a current measure is available).

## 5.6. Longitude

longitude sensor control interface, available for instance in the Yocto-GPS

The `YLongitude` class allows you to read and configure Yoctopuce longitude sensors. It inherits from `YSensor` class the core functions to read measurements, to register callback functions, and to access the autonomous datalogger.

### logicalName

Character string containing the logical name of the longitude sensor, initially empty. This attribute can be modified at will by the user. Once initialized to an non-empty value, it can be used to access the longitude sensor directly. If two longitude sensors with the same logical name are used in the same project, there is no way to determine which one answers when one tries accessing by logical name. The logical name is limited to 19 characters among `A..Z`,`a..z`,`0..9`,`_`, and `-`.

### advertisedValue

Short character string summarizing the current state of the longitude sensor, that is automatically advertised up to the parent hub. For a longitude sensor, the advertised value is the current value of the longitude.

### unit

Short character string representing the measuring unit for the longitude.

### currentValue

Current value of the longitude, in deg/1000, as a floating point number.

### lowestValue

Minimal value of the longitude, in deg/1000, as a floating point number.

### highestValue

Maximal value of the longitude, in deg/1000, as a floating point number.

### currentRawValue

Uncalibrated, unrounded raw value returned by the sensor, as a floating point number.

### logFrequency

Datalogger recording frequency, or "OFF" when measures should not be stored in the data logger flash memory.

### reportFrequency

Timed value notification frequency, or "OFF" when timed value notifications are disabled for this function.

### advMode

Measuring mode for the advertised value pushed to the parent hub.

### calibrationParam

Extra calibration parameters (for instance to compensate for the effects of an enclosure), as an array of 16 bit words.

### resolution

Measure resolution (i.e. precision of the numeric representation, not necessarily of the measure itself).

### sensorState

Sensor health state (zero when a current measure is available).

## 5.7. Altitude

altimeter control interface, available for instance in the Yocto-Altimeter-V2 or the Yocto-GPS

The `YAltitude` class allows you to read and configure Yoctopuce altimeters. It inherits from `YSensor` class the core functions to read measurements, to register callback functions, and to access the autonomous datalogger. This class adds the ability to configure the barometric pressure adjusted to sea level (QNH) for barometric sensors.

### logicalName

Character string containing the logical name of the altimeter, initially empty. This attribute can be modified at will by the user. Once initialized to an non-empty value, it can be used to access the altimeter directly. If two altimeters with the same logical name are used in the same project, there is no way to determine which one answers when one tries accessing by logical name. The logical name is limited to 19 characters among `A..Z`,`a..z`,`0..9`,`_`, and `-`.

### advertisedValue

Short character string summarizing the current state of the altimeter, that is automatically advertised up to the parent hub. For an altimeter, the advertised value is the current value of the altitude.

### unit

Short character string representing the measuring unit for the altitude.

### currentValue

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

### lowestValue

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

### highestValue

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

### currentRawValue

Uncalibrated, unrounded raw value returned by the sensor, as a floating point number.

### logFrequency

Datalogger recording frequency, or "OFF" when measures should not be stored in the data logger flash memory.

### reportFrequency

Timed value notification frequency, or "OFF" when timed value notifications are disabled for this function.

### advMode

Measuring mode for the advertised value pushed to the parent hub.

### calibrationParam

Extra calibration parameters (for instance to compensate for the effects of an enclosure), as an array of 16 bit words.

### resolution

Measure resolution (i.e. precision of the numeric representation, not necessarily of the measure itself).

### sensorState

Sensor health state (zero when a current measure is available).

### qnh

Barometric pressure adjusted to sea level (barometric altimeters only).

### technology

Sensor technology

## 5.8. GroundSpeed

ground speed sensor control interface, available for instance in the Yocto-GPS

The `YGroundSpeed` class allows you to read and configure Yoctopuce ground speed sensors. It inherits from `YSensor` class the core functions to read measurements, to register callback functions, and to access the autonomous datalogger.

### logicalName

Character string containing the logical name of the ground speed sensor, initially empty. This attribute can be modified at will by the user. Once initialized to an non-empty value, it can be used to access the ground speed sensor directly. If two ground speed sensors with the same logical name are used in the same project, there is no way to determine which one answers when one tries accessing by logical name. The logical name is limited to 19 characters among A..Z,a..z,0..9,_, and -.

### advertisedValue

Short character string summarizing the current state of the ground speed sensor, that is automatically advertised up to the parent hub. For a ground speed sensor, the advertised value is the current value of the ground speed.

### unit

Short character string representing the measuring unit for the ground speed.

### currentValue

Current value of the ground speed, in km/h, as a floating point number.

### lowestValue

Minimal value of the ground speed, in km/h, as a floating point number.

### highestValue

Maximal value of the ground speed, in km/h, as a floating point number.

### currentRawValue

Uncalibrated, unrounded raw value returned by the sensor, as a floating point number.

### logFrequency

Datalogger recording frequency, or "OFF" when measures should not be stored in the data logger flash memory.

### reportFrequency

Timed value notification frequency, or "OFF" when timed value notifications are disabled for this function.

### advMode

Measuring mode for the advertised value pushed to the parent hub.

### calibrationParam

Extra calibration parameters (for instance to compensate for the effects of an enclosure), as an array of 16 bit words.

### resolution

Measure resolution (i.e. precision of the numeric representation, not necessarily of the measure itself).

### sensorState

Sensor health state (zero when a current measure is available).

## 5.9. Gps

Geolocalization control interface (GPS, GNSS, ...), available for instance in the Yocto-GPS

The YGps class allows you to retrieve positioning data from a GPS/GNSS sensor. This class can provides complete positioning information. However, if you wish to define callbacks on position changes or record the position in the datalogger, you should use the YLatitude et YLongitude classes.

### logicalName

Character string containing the logical name of the geolocalization module, initially empty. This attribute can be modified at will by the user. Once initialized to an non-empty value, it can be used to access the geolocalization module directly. If two geolocalization modules with the same logical name are used in the same project, there is no way to determine which one answers when one tries accessing by logical name. The logical name is limited to 19 characters among A..Z,a..z,0..9,_, and -.

### advertisedValue

Short character string summarizing the current GPS reception state.

### isFixed

Indicates whether the receiver has found enough satellites to work

### satCount

Satellite count used to compute GPS position .

### satPerConst

Visible satellite count per constellation

### gpsRefreshRate

GPS receiver current refresh rate.

### coordSystem

Representation system to use for positioning data

### constellation

Satellite constellations to use for positioning computation

### latitude

Current latitude

### longitude

Current longitude

### dilution

Current dilution of precision

### altitude

Current altitude

### groundSpeed

Current ground speed

### direction

Current direction

### unixTime

Unix form of the current time (number of elapsed seconds since Jan 1970 1st)

### dateTime

Current time in the form "YYYY/MM/DD hh:mm:ss"

### utcOffset

Number of seconds between current time and UTC time (time zone)

### command

Magic attribute used to send commands to the device. If a command is not interpreted as expected, check the device logs.

## 5.10. DataLogger

DataLogger control interface, available on most Yoctopuce sensors.

A non-volatile memory for storing ongoing measured data is available on most Yoctopuce sensors. Recording can happen automatically, without requiring a permanent connection to a computer. The `YDataLogger` class controls the global parameters of the internal data logger. Recording control (start/stop) as well as data retreival is done at sensor objects level.

### logicalName

Character string containing the logical name of the data logger, initially empty. This attribute can be modified at will by the user. Once initialized to an non-empty value, it can be used to access the data logger directly. If two data loggers with the same logical name are used in the same project, there is no way to determine which one answers when one tries accessing by logical name. The logical name is limited to 19 characters among `A..Z`,`a..z`,`0..9`,`_`, and `-`.

### advertisedValue

Short character string summarizing the current state of the data logger, that is automatically advertised up to the parent hub. For a data logger, the advertised value is its recording state (ON or OFF).

### currentRunIndex

Current run number, corresponding to the number of time the module was powered on with the dataLogger enabled at some point.

### timeUTC

Current UTC time, in case it is desirable to bind an absolute time reference to the data stored by the data logger. This time must be set up by software.

### recording

Activation state of the data logger. The data logger can be enabled and disabled at will, using this attribute, but its state on power on is determined by the **autoStart** persistent attribute. When the datalogger is enabled but not yet ready to record data, its state is set to PENDING.

### autoStart

Automatic start of the data logger on power on. Setting this attribute ensures that the data logger is always turned on when the device is powered up, without need for a software command. Note: if the device doesn't have any time source at his disposal, it will wait for ~8 seconds before automatically starting to record.

### beaconDriven

Synchronize the sate of the localization beacon and the state of the data logger. If this attribute is set, it is possible to start the recording with the Yocto-button or the attribute `beacon` of the function YModule. In the same way, if the attribute `recording` is changed, the sate of the localization beacon is updated. Note: when this attribute is set the localization beacon pulses slower than usual.

### usage

Percentage of datalogger memory in use.

### clearHistory

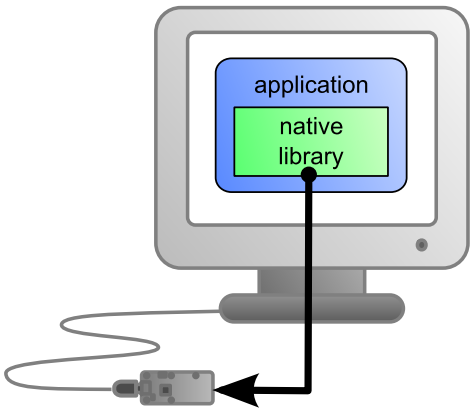
Attribute that can be set to true to clear recorded data.

## 5.11. What interface: Native, DLL or Service ?

There are several methods to control you Yoctopuce module by software.

### Native control

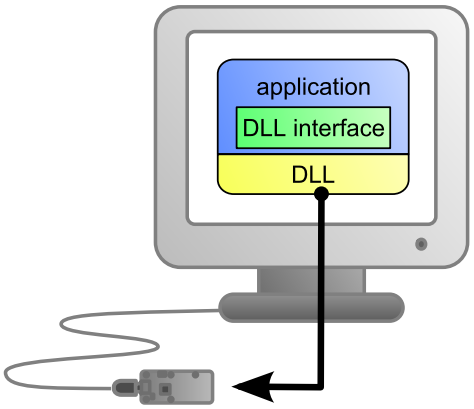
In this case, the software driving your project is compiled directly with a library which provides control of the modules. Objectively, it is the simplest and most elegant solution for the end user. The end user then only needs to plug the USB cable and run your software for everything to work. Unfortunately, this method is not always available or even possible.




*The application uses the native library to control the locally connected module*

### Native control by DLL

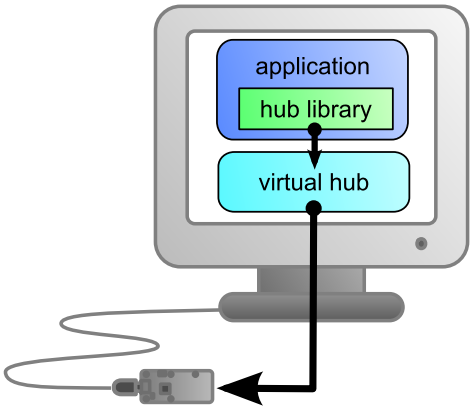
Here, the main part of the code controlling the modules is located in a DLL. The software is compiled with a small library which provides control of the DLL. It is the fastest method to code module support in a given language. Indeed, the "useful" part of the control code is located in the DLL which is the same for all languages: the effort to support a new language is limited to coding the small library which controls the DLL. From the end user stand point, there are few differences: one must simply make sure that the DLL is installed on the end user's computer at the same time as the main software.




*The application uses the DLL to natively control the locally connected module*

## Control by service

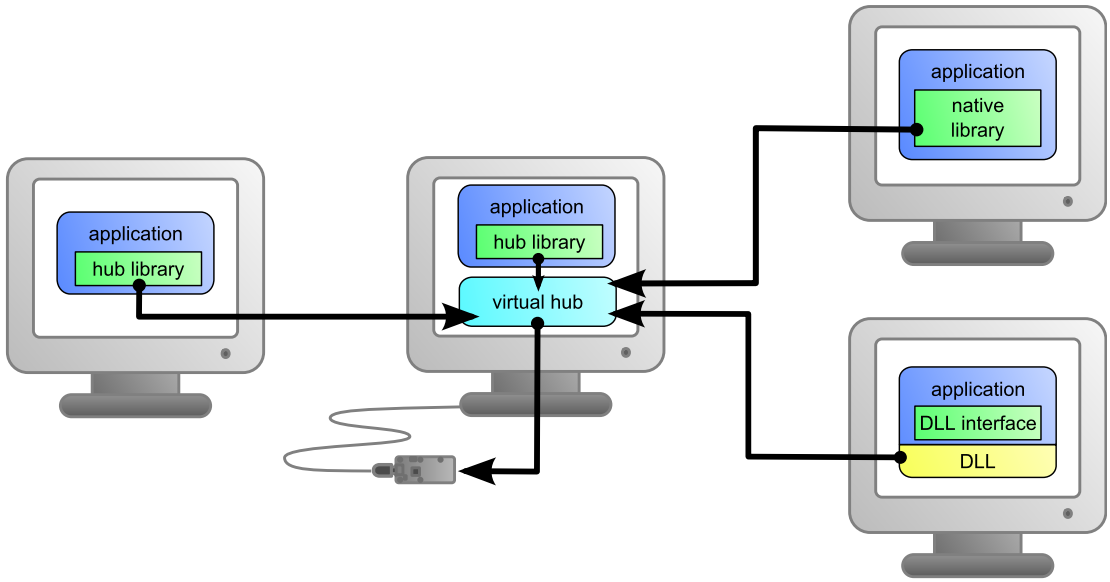
Some languages do simply not allow you to easily gain access to the hardware layers of the machine. It is the case for Javascript, for instance. To deal with this case, Yoctopuce provides a solution in the form of a small piece of software called *VirtualHub*[1]. It can access the modules, and your application only needs to use a library which offers all necessary functions to control the modules via this VirtualHub. The end users will have to start the VirtualHub before running the project control software itself, unless they decide to install the hub as a service/deamon, in which case the VirtualHub starts automatically when the machine starts up.




*The application connects itself to the VirtualHub to gain access to the module*

The service control method comes with a non-negligible advantage: the application does not need to run on the machine on which the modules are connected. The application can very well be located on another machine which connects itself to the service to drive the modules. Moreover, the native libraries and DLL mentioned above are also able to connect themselves remotely to one or several machines running VirtualHub.

[1] www.yoctopuce.com/EN/virtualhub.php




*When a VirtualHub is used, the control application does not need to reside on the same machine as the module.*

Whatever the selected programming language and the control paradigm used, programming itself stays strictly identical. From one language to another, functions bear exactly the same name, and have the same parameters. The only differences are linked to the constraints of the languages themselves.

| Language | Native | Native with DLL | VirtualHub |
|---|---|---|---|
| C++ | ✔ | ✔ | ✔ |
| Objective-C | ✔ | - | ✔ |
| Delphi | - | ✔ | ✔ |
| Python | - | ✔ | ✔ |
| VisualBasic .Net | - | ✔ | ✔ |
| C# .Net | - | ✔ | ✔ |
| C# UWP | ✔ | - | ✔ |
| EcmaScript / JavaScript | - | - | ✔ |
| PHP | - | - | ✔ |
| Java | - | ✔ | ✔ |
| Java for Android | ✔ | - | ✔ |
| Command line | ✔ | - | ✔ |
| LabVIEW | - | ✔ | ✔ |

*Support methods for different languages*

### Limitations of the Yoctopuce libraries

Natives et DLL libraries have a technical limitation. On the same computer, you cannot concurrently run several applications accessing Yoctopuce devices directly. If you want to run several projects on the same computer, make sure your control applications use Yoctopuce devices through a *VirtualHub* software. The modification is trivial: it is just a matter of parameter change in the `yRegisterHub()` call.

## 5.12. Programming, where to start?

At this point of the user's guide, you should know the main theoretical points of your Yocto-GPS. It is now time to practice. You must download the Yoctopuce library for your favorite programming language from the Yoctopuce web site[2]. Then skip directly to the chapter corresponding to the chosen programming language.

All the examples described in this guide are available in the programming libraries. For some languages, the libraries also include some complete graphical applications, with their source code.

[2] http://www.yoctopuce.com/EN/libraries.php

When you have mastered the basic programming of your module, you can turn to the chapter on advanced programming that describes some techniques that will help you make the most of your Yocto-GPS.

# 6. Using the Yocto-GPS in command line

When you want to perform a punctual operation on your Yocto-GPS, 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].

## 6.1. 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-GPS, open a shell, and start working by typing for example:

```
C:\>YGps any get_latitude
```

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.

## 6.2. 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.

## 6.3. Control of the Latitude function

To control the Latitude function of your Yocto-GPS, you need the YLatitude executable file.

For instance, you can launch:

```
C:\>YGps any get_latitude
```

This example uses the "*any*" target to indicate that we want to work on the first Latitude 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-GPS module with the *YGNSSMK1-123456* serial number which you have called "*MyModule*", and its latitude 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:\>YLatitude YGNSSMK1-123456.latitude describe

C:\>YLatitude YGNSSMK1-123456.MyFunction describe

C:\>YLatitude MyModule.latitude describe

C:\>YLatitude MyModule.MyFunction describe

C:\>YLatitude MyFunction describe
```

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

```
C:\>YLatitude all describe
```

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

```
C:\>YLatitude /help
```

## 6.4. 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 YGNSSMK1-12346 set_logicalName MonPremierModule

C:\>YModule YGNSSMK1-12346 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 YGNSSMK1-12346 set_logicalName MonPremierModule
C:\>YModule YGNSSMK1-12346 saveToFlash
```

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

```
C:\>YModule -s  YGNSSMK1-12346 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.

## 6.5. 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

# 7. Using Yocto-GPS with JavaScript / EcmaScript

EcmaScript is the official name of the standardized version of the web-oriented programming language commonly referred to as *JavaScript*. This Yoctopuce library take advantages of advanced features introduced in EcmaScript 2017. It has therefore been named *Library for JavaScript / EcmaScript 2017* to differentiate it from the previous *Library for JavaScript*, now deprecated in favor of this new version.

This library provides access to Yoctopuce devices for modern JavaScript engines. It can be used within a browser as well as with Node.js. The library will automatically detect upon initialization whether the runtime environment is a browser or a Node.js virtual machine, and use the most appropriate system libraries accordingly.

Asynchronous communication with the devices is handled across the whole library using Promise objects, leveraging the new EcmaScript 2017 `async` / `await` non-blocking syntax for asynchronous I/O (see below). This syntax is now available out-of-the-box in most Javascript engines. No transpilation is needed: no Babel, no jspm, just plain Javascript. Here is your favorite engines minimum version needed to run this code. All of them are officially released at the time we write this document.

- Node.js v7.6 and later
- Firefox 52
- Opera 42 (incl. Android version)
- Chrome 55 (incl. Android version)
- Safari 10.1 (incl. iOS version)
- Android WebView 55
- Google V8 Javascript engine v5.5

If you need backward-compatibility with older releases, you can always run Babel to transpile your code and the library to older standards, as described a few paragraphs below.

We don't suggest using `jspm 0.17` anymore since that tool is still in Beta after 18 month, and having to use an extra tool to implement our library is pointless now that `async` / `await` are part of the standard.

## 7.1. Blocking I/O versus Asynchronous I/O in JavaScript

JavaScript is single-threaded by design. That means, if a program is actively waiting for the result of a network-based operation such as reading from a sensor, the whole program is blocked. In browser environments, this can even completely freeze the user interface. For this reason, the use of blocking I/O in JavaScript is strongly discouraged nowadays, and blocking network APIs are getting deprecated everywhere.

Instead of using parallel threads, JavaScript relies on asynchronous I/O to handle operations with a possible long timeout: whenever a long I/O call needs to be performed, it is only triggered and but then the code execution flow is terminated. The JavaScript engine is therefore free to handle other pending tasks, such as UI. Whenever the pending I/O call is completed, the system invokes a callback function with the result of the I/O call to resume execution of the original execution flow.

When used with plain callback functions, as pervasive in Node.js libraries, asynchronous I/O tend to produce code with poor readability, as the execution flow is broken into many disconnected callback functions. Fortunately, new methods have emerged recently to improve that situation. In particular, the use of *Promise* objects to abstract and work with asynchronous tasks helps a lot. Any function that makes a long I/O operation can return a *Promise*, which can be used by the caller to chain subsequent operations in the same flow. Promises are part of EcmaScript 2015 standard.

Promise objects are good, but what makes them even better is the new `async` / `await` keywords to handle asynchronous I/O:

- a function declared *async* will automatically encapsulate its result as a Promise
- within an *async* function, any function call prefixed with by *await* will chain the Promise returned by the function with a promise to resume execution of the caller
- any exception during the execution of an *async* function will automatically invoke the Promise failure continuation

Long story made short, *async* and *await* make it possible to write EcmaScript code with all benefits of asynchronous I/O, but without breaking the code flow. It is almost like multi-threaded execution, except that control switch between pending tasks only happens at places where the *await* keyword appears.

We have therefore chosen to write our new EcmaScript library using Promises and *async* functions, so that you can use the friendly *await* syntax. To keep it easy to remember, **all public methods** of the EcmaScript library **are *async***, i.e. return a Promise object, **except**:

- `GetTickCount()`, because returning a time stamp asynchronously does not make sense...
- `FindModule()`, `FirstModule()`, `nextModule()`,... because device detection and enumeration always work on internal device lists handled in background, and does not require immediate asynchronous I/O.

# 7.2. Using Yoctopuce library for JavaScript / EcmaScript 2017

JavaScript is one of those languages which do not generally allow you to directly access the hardware layers of your computer. Therefore the library can only be used to access network-enabled devices (connected through a YoctoHub), or USB devices accessible through Yoctopuce TCP/IP to USB gateway, named *VirtualHub*.

Go to the Yoctopuce web site and download the following items:

- The Javascript / EcmaScript 2017 programming library[1]
- The VirtualHub software[2] for Windows, Mac OS X or Linux, depending on your OS

Extract the library files in a folder of your choice, you will find many of examples in it. Connect your modules and start the VirtualHub software. You do not need to install any driver.

## Using the official Yoctopuce library for node.js

Start by installing the latest Node.js version (v7.6 or later) on your system. It is very easy. You can download it from the official web site: http://nodejs.org. Make sure to install it fully, including npm, and add it to the system path.

[1] www.yoctopuce.com/EN/libraries.php
[2] www.yoctopuce.com/EN/virtualhub.php

To give it a try, go into one of the example directory (for instance example_nodejs/Doc-Inventory). You will see that it include an application description file (package.json) and a source file (demo.js). To download and setup the libraries needed by this example, just run:

```
npm install
```

Once done, you can start the example file using:

```
node demo.js
```

## Using a local copy of the Yoctopuce library with node.js

If for some reason you need to make changes to the Yoctopuce library, you can easily configure your project to use the local copy in the `lib/` subdirectory rather than the official npm package. In order to do so, simply type the following command in your project directory:

```
npm link ../../lib
```

## Using the Yoctopuce library within a browser (HTML)

For HTML examples, it is even simpler: there is nothing to install. Each example is a single HTML file that you can open in a browser to try it. In this context, loading the Yoctopuce library is no different from any standard HTML script include tag.

## Using the Yoctoluce library on older JavaScript engines

If you need to run this library on older JavaScript engines, you can use Babel[3] to transpile your code and the library into older JavaScript standards. To install Babel with typical settings, simply use:

```
npm instal -g babel-cli
npm instal babel-preset-env
```

You would typically ask Babel to put the transpiled files in another directory, named `compat` for instance. Your files and all files of the Yoctopuce library should be transpiled, as follow:

```
babel --presets env demo.js --out-dir compat/
babel --presets env ../../lib --out-dir compat/
```

Although this approach is based on node.js toolchain, it actually works as well for transpiling JavaScript files for use in a browser. The only thing that you cannot do so easily is transpiling JavaScript code embedded directly in an HTML page. You have to use an external script file for using EcmaScript 2017 syntax with Babel.

Babel has many smart features, such as a watch mode that will automatically refresh transpiled files whenever the source file is changed, but this is beyond the scope of this note. You will find more in Babel documentation.

## Backward-compatibility with the old JavaScript library

This new library is not fully backward-compatible with the old JavaScript library, because there is no way to transparently map the old blocking API to the new asynchronous API. The method names however are the same, and old synchronous code can easily be made asynchronous just by adding the proper `await` keywords before the method calls. For instance, simply replace:

```
beaconState = module.get_beacon();
```

by

[3] http://babeljs.io

```
beaconState = await module.get_beacon();
```

Apart from a few exceptions, most `XXX_async` redundant methods have been removed as well, as they would have introduced confusion on the proper way of handling asynchronous behaviors. It is however very simple to get an *async* method to invoke a callback upon completion, using the returned Promise object. For instance, you can replace:

```
module.get_beacon_async(callback, myContext);
```

by

```
module.get_beacon().then(function(res) { callback(myContext, module, res); });
```

In some cases, it might be desirable to get a sensor value using a method identical to the old synchronous methods (without using Promises), even if it returns a slightly outdated cached value since I/O is not possible. For this purpose, the EcmaScript library introduce new classes called *synchronous proxies*. A synchronous proxy is an object that mirrors the most recent state of the connected class, but can be read using regular synchronous function calls. For instance, instead of writing:

```
async function logInfo(module)
{
    console.log('Name: '+await module.get_logicalName());
    console.log('Beacon: '+await module.get_beacon());
}

...
logInfo(myModule);
...
```

you can use:

```
function logInfoProxy(moduleSyncProxy)
{
    console.log('Name: '+moduleProxy.get_logicalName());
    console.log('Beacon: '+moduleProxy.get_beacon());
}

logInfoSync(await myModule.get_syncProxy());
```

You can also rewrite this last asynchronous call as:

```
myModule.get_syncProxy().then(logInfoProxy);
```

## 7.3. Control of the Latitude function

A few lines of code are enough to use a Yocto-GPS. Here is the skeleton of a JavaScript code snipplet to use the Latitude function.

```
// For Node.js, we use function require()
// For HTML, we would use &lt;script src="..."&gt;
require('yoctolib-es2017/yocto_api.js');
require('yoctolib-es2017/yocto_latitude.js');

[...]
// Get access to your device, through the VirtualHub running locally
await YAPI.RegisterHub('127.0.0.1');
[...]

// Retrieve the object used to interact with the device
var latitude = YLatitude.FindLatitude("YGNSSMK1-123456.latitude");

// Check that the module is online to handle hot-plug
```

```
if(await latitude.isOnline())
{
    // Use latitude.get_currentValue()
    [...]
}
```

Let us look at these lines in more details.

## yocto_api and yocto_latitude import

These two import provide access to functions allowing you to manage Yoctopuce modules. `yocto_api` is always needed, `yocto_latitude` is necessary to manage modules containing a latitude sensor, such as Yocto-GPS. Other imports can be useful in other cases, such as `YModule` which can let you enumerate any type of Yoctopuce device.

## YAPI.RegisterHub

The `RegisterHub` method allows you to indicate on which machine the Yoctopuce modules are located, more precisely on which machine the VirtualHub software is running. In our case, the `127.0.0.1:4444` address indicates the local machine, port `4444` (the standard port used by Yoctopuce). You can very well modify this address, and enter the address of another machine on which the VirtualHub software is running, or of a YoctoHub. If the host cannot be reached, this function will trigger an exception.

## YLatitude.FindLatitude

The `FindLatitude` method allows you to find a latitude sensor from the serial number of the module on which it resides and from its function name. You can also use logical names, as long as you have initialized them. Let us imagine a Yocto-GPS module with serial number *YGNSSMK1-123456* which you have named "*MyModule*", and for which you have given the *latitude* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
latitude = YLatitude.FindLatitude("YGNSSMK1-123456.latitude")
latitude = YLatitude.FindLatitude("YGNSSMK1-123456.MaFonction")
latitude = YLatitude.FindLatitude("MonModule.latitude")
latitude = YLatitude.FindLatitude("MonModule.MaFonction")
latitude = YLatitude.FindLatitude("MaFonction")
```

`YLatitude.FindLatitude` returns an object which you can then use at will to control the latitude sensor.

## isOnline

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

## get_latitude

The `get_latitude()` method of the object returned by `YGps.FindGps` provides the latitude currently measured by the Yocto-GPS. The value returned is a string, the format will vary according to the Yocto-GPS configuration. To get a floating point value, no matter the configuartion, use the YLatitude Class.

## A real example, for Node.js

Open a command window (a terminal, a shell...) and go into the directory **example_nodejs/Doc-GettingStarted-Yocto-GPS** within Yoctopuce library for JavaScript / EcmaScript 2017. In there, you will find a file named `demo.js` with the sample code below, which uses the functions explained above, but this time used with all side materials needed to make it work nicely as a small demo.

If your Yocto-GPS is not connected on the host running the browser, replace in the example the address `127.0.0.1` by the IP address of the host on which the Yocto-GPS is connected and where you run the VirtualHub.

```
"use strict";

require('yoctolib-es2017/yocto_api.js');
require('yoctolib-es2017/yocto_gps.js');

let gps;

async function startDemo()
{
    await YAPI.LogUnhandledPromiseRejections();
    await YAPI.DisableExceptions();

    // Setup the API to use the VirtualHub on local machine
    let errmsg = new YErrorMsg();
    if(await YAPI.RegisterHub('127.0.0.1', errmsg) != YAPI.SUCCESS) {
        console.log('Cannot contact VirtualHub on 127.0.0.1: '+errmsg.msg);
        return;
    }

    // Select specified device, or use first available one
    let serial = process.argv[process.argv.length-1];
    if(serial[8] != '-') {
        // by default use any connected module suitable for the demo
        let anysensor = YGps.FirstGps();
        if(anysensor) {
            let module = await anysensor.module();
            serial = await module.get_serialNumber();
        } else {
            console.log('No matching sensor connected, check cable !');
            return;
        }
    }
    console.log('Using device '+serial);
    gps = YGps.FindGps(serial+".gps");

    refresh();
}

async function refresh()
{
    if (!await gps.isOnline()) {
        console.log('Module not connected');
    } else if (await gps.get_isFixed() != YGps.ISFIXED_TRUE) {
        console.log('fixing...');
    } else {
        console.log('Position : '+(await gps.get_latitude())
            + ' ' + (await gps.get_longitude()));
    }
    setTimeout(refresh, 500);
}

startDemo();
```

As explained at the beginning of this chapter, you need to have Node.js v7.6 or later installed to try this example. When done, you can type the following two commands to automatically download and install the dependencies for building this example:

```
npm install
```

You can the start the sample code within Node.js using the following command, replacing the [...] by the arguments that you want to pass to the demo code:

```
node demo.js [...]
```

## Same example, but this time running in a browser

If you want to see how to use the library within a browser rather than with Node.js, switch to the directory **example_html/Doc-GettingStarted-Yocto-GPS**. You will find there a single HTML file, with a JavaScript section similar to the code above, but with a few changes since it has to interact through an HTML page rather than through the JavaScript console.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Hello World</title>
  <script src="../../lib/yocto_api.js"></script>
  <script src="../../lib/yocto_gps.js"></script>
  <script>
    async function startDemo()
    {
      await YAPI.LogUnhandledPromiseRejections();
      await YAPI.DisableExceptions();

      // Setup the API to use the VirtualHub on local machine
      let errmsg = new YErrorMsg();
      if(await YAPI.RegisterHub('127.0.0.1', errmsg) != YAPI.SUCCESS) {
      alert('Cannot contact VirtualHub on 127.0.0.1: '+errmsg.msg);
    }
      refresh();
    }

    async function refresh()
    {
      let serial = document.getElementById('serial').value;
      if(serial == '') {
        // by default use any connected module suitable for the demo
        let anysensor = YGps.FirstGps();
        if(anysensor) {
          let module = await anysensor.module();
          serial = await module.get_serialNumber();
          document.getElementById('serial').value = serial;
        }
      }
      let gps = YGps.FindGps(serial+".gps");

      if (await gps.isOnline()) {
      document.getElementById('msg').value = '';
      if (await gps.get_isFixed() != YGps.ISFIXED_TRUE) {
        document.getElementById("gps").value = 'fixing...';
      } else {
        document.getElementById("gps").value = (await gps.get_latitude()) + ' ' + (await
gps.get_longitude());
      }
    } else {
      document.getElementById('msg').value = 'Module not connected';
    }
      setTimeout(refresh, 500);
    }

    startDemo();
  </script>
</head>
<body>
Module to use: <input id='serial'>
<input id='msg' style='color:red;border:none;' readonly><br>
Current position :  <input id='gps' readonly><br>
</body>
</html>
```

No installation is needed to run this example, all you have to do is open the HTML file using a web browser,

## 7.4. Control of the module part

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

```
"use strict";

require('yoctolib-es2017/yocto_api.js');

async function startDemo(args)
{
    await YAPI.LogUnhandledPromiseRejections();
```

```
    // Setup the API to use the VirtualHub on local machine
    let errmsg = new YErrorMsg();
    if(await YAPI.RegisterHub('127.0.0.1', errmsg) != YAPI.SUCCESS) {
        console.log('Cannot contact VirtualHub on 127.0.0.1: '+errmsg.msg);
        return;
    }

    // Select the relay to use
    let module = YModule.FindModule(args[0]);
    if(await module.isOnline()) {
        if(args.length > 1) {
            if(args[1] == 'ON') {
                await module.set_beacon(YModule.BEACON_ON);
            } else {
                await module.set_beacon(YModule.BEACON_OFF);
            }
        }
        console.log('serial:       '+await module.get_serialNumber());
        console.log('logical name: '+await module.get_logicalName());
        console.log('luminosity:   '+await module.get_luminosity()+'%');
        console.log('beacon:       '+(await module.get_beacon()==YModule.BEACON_ON
?'ON':'OFF'));
        console.log('upTime:       '+parseInt(await module.get_upTime()/1000)+' sec');
        console.log('USB current:  '+await module.get_usbCurrent()+' mA');
        console.log('logs:');
        console.log(await module.get_lastLogs());
    } else {
        console.log("Module not connected (check identification and USB cable)\n");
    }
    await YAPI.FreeAPI();
}

if(process.argv.length < 2) {
    console.log("usage: node demo.js <serial or logicalname> [ ON | OFF ]");
} else {
    startDemo(process.argv.slice(2));
}
```

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

## Changing the module settings

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

```
"use strict";

require('yoctolib-es2017/yocto_api.js');

async function startDemo(args)
{
    await YAPI.LogUnhandledPromiseRejections();

    // Setup the API to use the VirtualHub on local machine
    let errmsg = new YErrorMsg();
    if(await YAPI.RegisterHub('127.0.0.1', errmsg) != YAPI.SUCCESS) {
        console.log('Cannot contact VirtualHub on 127.0.0.1: '+errmsg.msg);
        return;
    }

    // Select the relay to use
    let module = YModule.FindModule(args[0]);
    if(await module.isOnline()) {
        if(args.length > 1) {
            let newname = args[1];
            if (!await YAPI.CheckLogicalName(newname)) {
```

```
                console.log("Invalid name (" + newname + ")");
                process.exit(1);
            }
            await module.set_logicalName(newname);
            await module.saveToFlash();
        }
        console.log('Current name: '+await module.get_logicalName());
    } else {
        console.log("Module not connected (check identification and USB cable)\n");
    }
    await YAPI.FreeAPI();
}

if(process.argv.length < 2) {
    console.log("usage: node demo.js <serial> [newLogicalName]");
} else {
    startDemo(process.argv.slice(2));
}
```

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

## Listing the modules

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

```
"use strict";

require('yoctolib-es2017/yocto_api.js');

async function startDemo()
{
    await YAPI.LogUnhandledPromiseRejections();
    await YAPI.DisableExceptions();

    // Setup the API to use the VirtualHub on local machine
    let errmsg = new YErrorMsg();
    if (await YAPI.RegisterHub('127.0.0.1', errmsg) != YAPI.SUCCESS) {
        console.log('Cannot contact VirtualHub on 127.0.0.1');
        return;
    }
    refresh();
}

async function refresh()
{
    try {
        let errmsg = new YErrorMsg();
        await YAPI.UpdateDeviceList(errmsg);

        let module = YModule.FirstModule();
        while(module) {
            let line = await module.get_serialNumber();
            line += '(' + (await module.get_productName()) + ')';
            console.log(line);
            module = module.nextModule();
        }
        setTimeout(refresh, 500);
    } catch(e) {
        console.log(e);
    }
}

try {
    startDemo();
} catch(e) {
    console.log(e);
```

```
}
```

## 7.5. Error handling

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

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

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

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

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

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

# 8. Using Yocto-GPS with PHP

PHP is, like Javascript, an atypical language when interfacing with hardware is at stakes. Nevertheless, using PHP with Yoctopuce modules provides you with the opportunity to very easily create web sites which are able to interact with their physical environment, and this is not available to every web server. This technique has a direct application in home automation: a few Yoctopuce modules, a PHP server, and you can interact with your home from anywhere on the planet, as long as you have an internet connection.

PHP is one of those languages which do not allow you to directly access the hardware layers of your computer. Therefore you need to run a virtual hub on the machine on which your modules are connected.

To start your tests with PHP, you need a PHP 5.3 (or more) server[1], preferably locally on you machine. If you wish to use the PHP server of your internet provider, it is possible, but you will probably need to configure your ADSL router for it to accept and forward TCP request on the 4444 port.

## 8.1. Getting ready

Go to the Yoctopuce web site and download the following items:

- The PHP programming library[2]
- The VirtualHub software[3] for Windows, Mac OS X, or Linux, depending on your OS

Decompress the library files in a folder of your choice accessible to your web server, connect your modules, run the VirtualHub software, and you are ready to start your first tests. You do not need to install any driver.

## 8.2. Control of the Latitude function

A few lines of code are enough to use a Yocto-GPS. Here is the skeleton of a PHP code snipplet to use the Latitude function.

```
include('yocto_api.php');
include('yocto_latitude.php');
```

[1] A couple of free PHP servers: easyPHP for Windows, MAMP for Mac OS X.
[2] www.yoctopuce.com/EN/libraries.php
[3] www.yoctopuce.com/EN/virtualhub.php

```
[...]
// Get access to your device, through the VirtualHub running locally
YAPI::RegisterHub('http://127.0.0.1:4444/',$errmsg);
[...]

// Retrieve the object used to interact with the device
$latitude = YLatitude::FindLatitude("YGNSSMK1-123456.latitude");

// Check that the module is online to handle hot-plug
if($latitude->isOnline())
{
    // Use $latitude->get_currentValue()
    [...]
}
```

Let's look at these lines in more details.

## yocto_api.php and yocto_latitude.php

These two PHP includes provides access to the functions allowing you to manage Yoctopuce modules. `yocto_api.php` must always be included, `yocto_latitude.php` is necessary to manage modules containing a latitude sensor, such as Yocto-GPS.

## YAPI::RegisterHub

The `YAPI::RegisterHub` function allows you to indicate on which machine the Yoctopuce modules are located, more precisely on which machine the VirtualHub software is running. In our case, the `127.0.0.1:4444` address indicates the local machine, port `4444` (the standard port used by Yoctopuce). You can very well modify this address, and enter the address of another machine on which the VirtualHub software is running.

## YLatitude::FindLatitude

The `YLatitude::FindLatitude` function allows you to find a latitude sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-GPS module with serial number *YGNSSMK1-123456* which you have named "*MyModule*", and for which you have given the *latitude* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
$latitude = YLatitude::FindLatitude("YGNSSMK1-123456.latitude");
$latitude = YLatitude::FindLatitude("YGNSSMK1-123456.MyFunction");
$latitude = YLatitude::FindLatitude("MyModule.latitude");
$latitude = YLatitude::FindLatitude("MyModule.MyFunction");
$latitude = YLatitude::FindLatitude("MyFunction");
```

`YLatitude::FindLatitude` returns an object which you can then use at will to control the latitude sensor.

## isOnline

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

## get_latitude

The `get_latitude()` method of the object returned by `yFindGps` provides the latitude currently measured by the Yocto-GPS. The value returned is a string, the format will vary according to the Yocto-GPS configuration. To get a floating point value, no matter the configuartion, use the YLatitude Class.

### A real example

Open your preferred text editor[4], copy the code sample below, save it with the Yoctopuce library files in a location which is accessible to you web server, then use your preferred web browser to access this page. The code is also provided in the directory **Examples/Doc-GettingStarted-Yocto-GPS** of the Yoctopuce library.

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

```
<HTML>
<HEAD>
 <TITLE>Hello World</TITLE>
</HEAD>
<BODY>
<?php
  include('yocto_api.php');
  include('yocto_gps.php');

  // Use explicit error handling rather than exceptions
  //YAPI::DisableExceptions();

  // Setup the API to use the VirtualHub on local machine
  if(YAPI::RegisterHub('http://127.0.0.1:4444/',$errmsg) != YAPI::SUCCESS) {
      die("Cannot contact VirtualHub on 127.0.0.1");
  }

  @$serial = $_GET['serial'];
  if ($serial != '') {
      // Check if a specified module is available online
      $gps = YGps::FindGps("$serial.gps");
      if (!$gps->isOnline()) {
          die("Module not connected (check serial and USB cable)");
      }
  } else {
      // or use any connected module suitable for the demo
      $gps = YGps::FirstGps();
      if(is_null($gps)) {
          die("No module connected (check USB cable)");
      } else {
          $serial = $gps->module()->get_serialnumber();
      }
  }
  Print("Module to use: <input name='serial' value='$serial'><br>");

  if  ($gps->get_isFixed()!=Y_ISFIXED_TRUE)
      Print("Gps : fixing...<br>");
  else
      Printf("Gps : %s %s<br>", $gps->get_latitude(), $gps->get_longitude());
  YAPI::FreeAPI();

  // trigger auto-refresh after one second
  Print("<script language='javascript1.5' type='text/JavaScript'>\n");
  Print("setTimeout('window.location.reload()',1000);");
  Print("</script>\n");
?>
</BODY>
</HTML>
```

## 8.3. Control of the module part

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

```
<HTML>
<HEAD>
 <TITLE>Module Control</TITLE>
</HEAD>
```

[4] If you do not have a text editor, use Notepad rather than Microsoft Word.

```
<BODY>
<FORM method='get'>
<?php
  include('yocto_api.php');

  // Use explicit error handling rather than exceptions
  YAPI::DisableExceptions();

  // Setup the API to use the VirtualHub on local machine
  if(YAPI::RegisterHub('http://127.0.0.1:4444/',$errmsg) != YAPI::SUCCESS) {
      die("Cannot contact VirtualHub on 127.0.0.1 : ".$errmsg);
  }

  @$serial = $_GET['serial'];
  if ($serial != '') {
      // Check if a specified module is available online
      $module = YModule::FindModule("$serial");
      if (!$module->isOnline()) {
          die("Module not connected (check serial and USB cable)");
      }
  } else {
      // or use any connected module suitable for the demo
      $module = YModule::FirstModule();
      if($module) { // skip VirtualHub
          $module = $module->nextModule();
      }
      if(is_null($module)) {
          die("No module connected (check USB cable)");
      } else {
          $serial = $module->get_serialnumber();
      }
  }
  Print("Module to use: <input name='serial' value='$serial'><br>");

  if (isset($_GET['beacon'])) {
      if ($_GET['beacon']=='ON')
          $module->set_beacon(Y_BEACON_ON);
      else
          $module->set_beacon(Y_BEACON_OFF);
  }
  printf('serial: %s<br>',$module->get_serialNumber());
  printf('logical name: %s<br>',$module->get_logicalName());
  printf('luminosity: %s<br>',$module->get_luminosity());
  print('beacon: ');
  if($module->get_beacon() == Y_BEACON_ON) {
      printf("<input type='radio' name='beacon' value='ON' checked>ON ");
      printf("<input type='radio' name='beacon' value='OFF'>OFF<br>");
  } else {
      printf("<input type='radio' name='beacon' value='ON'>ON ");
      printf("<input type='radio' name='beacon' value='OFF' checked>OFF<br>");
  }
  printf('upTime: %s sec<br>',intVal($module->get_upTime()/1000));
  printf('USB current: %smA<br>',$module->get_usbCurrent());
  printf('logs:<br><pre>%s</pre>',$module->get_lastLogs());
  YAPI::FreeAPI();
?>
<input type='submit' value='refresh'>
</FORM>
</BODY>
</HTML>
```

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

## Changing the module settings

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

```
<HTML>
<HEAD>
 <TITLE>save settings</TITLE>
<BODY>
<FORM method='get'>
<?php
  include('yocto_api.php');

  // Use explicit error handling rather than exceptions
  YAPI::DisableExceptions();

  // Setup the API to use the VirtualHub on local machine
  if(YAPI::RegisterHub('http://127.0.0.1:4444/',$errmsg) != YAPI::SUCCESS) {
      die("Cannot contact VirtualHub on 127.0.0.1");
  }

  @$serial = $_GET['serial'];
  if ($serial != '') {
      // Check if a specified module is available online
      $module = YModule::FindModule("$serial");
      if (!$module->isOnline()) {
          die("Module not connected (check serial and USB cable)");
      }
  } else {
      // or use any connected module suitable for the demo
      $module = YModule::FirstModule();
      if($module) { // skip VirtualHub
          $module = $module->nextModule();
      }
      if(is_null($module)) {
          die("No module connected (check USB cable)");
      } else {
          $serial = $module->get_serialnumber();
      }
  }
  Print("Module to use: <input name='serial' value='$serial'><br>");

  if (isset($_GET['newname'])){
      $newname = $_GET['newname'];
      if (!yCheckLogicalName($newname))
          die('Invalid name');
      $module->set_logicalName($newname);
      $module->saveToFlash();
  }
  printf("Current name: %s<br>", $module->get_logicalName());
  print("New name: <input name='newname' value='' maxlength=19><br>");
  YAPI::FreeAPI();
?>
<input type='submit'>
</FORM>
</BODY>
</HTML>
```

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

## Listing the modules

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

```
<HTML>
<HEAD>
 <TITLE>inventory</TITLE>
</HEAD>
<BODY>
<H1>Device list</H1>
<TT>
<?php
```

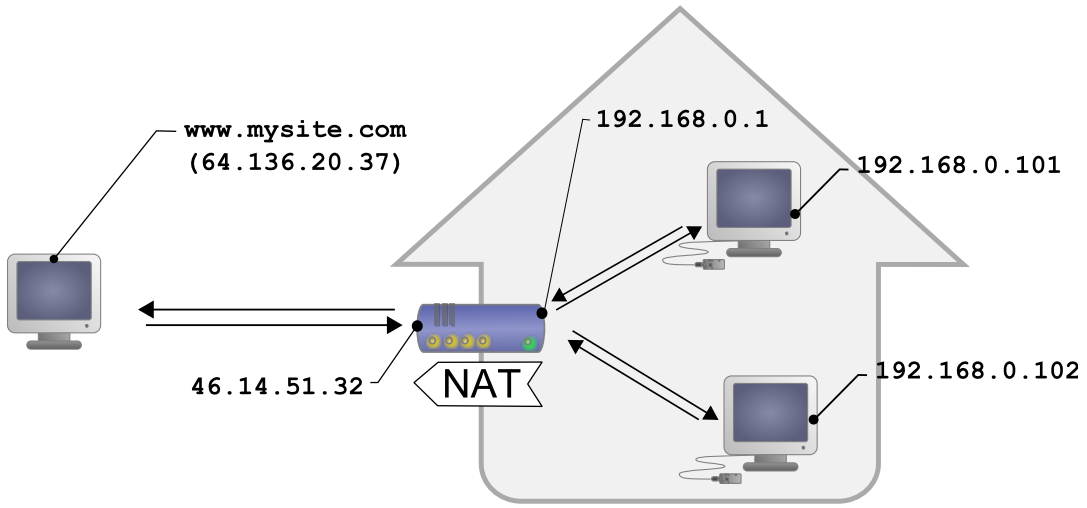
```
    include('yocto_api.php');
    YAPI::RegisterHub("http://127.0.0.1:4444/");
    $module   = YModule::FirstModule();
    while (!is_null($module)) {
        printf("%s (%s)<br>", $module->get_serialNumber(),
               $module->get_productName());
        $module=$module->nextModule();
    }
    YAPI::FreeAPI();
?>
</TT>
</BODY>
</HTML>
```

## 8.4. HTTP callback API and NAT filters

The PHP library is able to work in a specific mode called *HTTP callback Yocto-API*. With this mode, you can control Yoctopuce devices installed behind a NAT filter, such as a DSL router for example, and this without needing to open a port. The typical application is to control Yoctopuce devices, located on a private network, from a public web site.

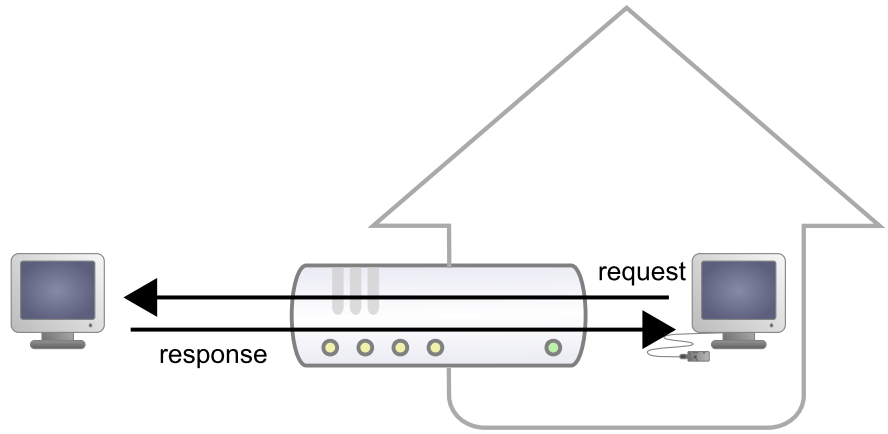
### The NAT filter: advantages and disadvantages

A DSL router which translates network addresses (NAT) works somewhat like a private phone switchboard (a PBX): internal extensions can call each other and call the outside; but seen from the outside, there is only one official phone number, that of the switchboard itself. You cannot reach the internal extensions from the outside.

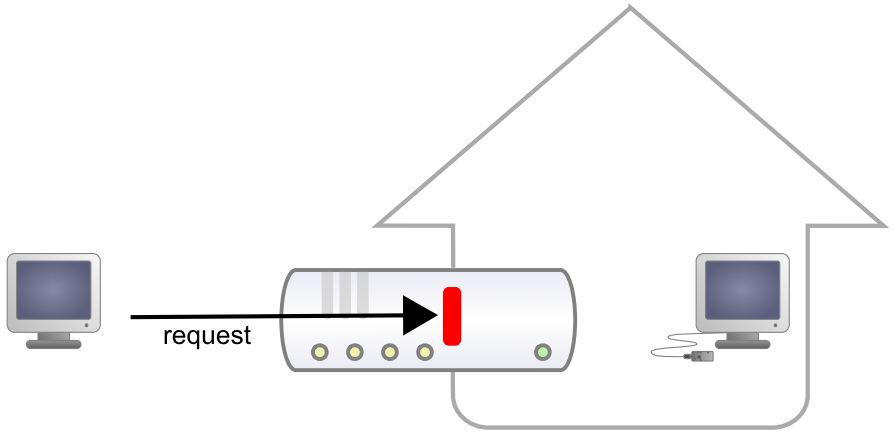



*Typical DSL configuration: LAN machines are isolated from the outside by the DSL router*

Transposed to the network, we have the following: appliances connected to your home automation network can communicate with one another using a local IP address (of the 192.168.xxx.yyy type), and contact Internet servers through their public address. However, seen from the outside, you have only one official IP address, assigned to the DSL router only, and you cannot reach your network appliances directly from the outside. It is rather restrictive, but it is a relatively efficient protection against intrusions.



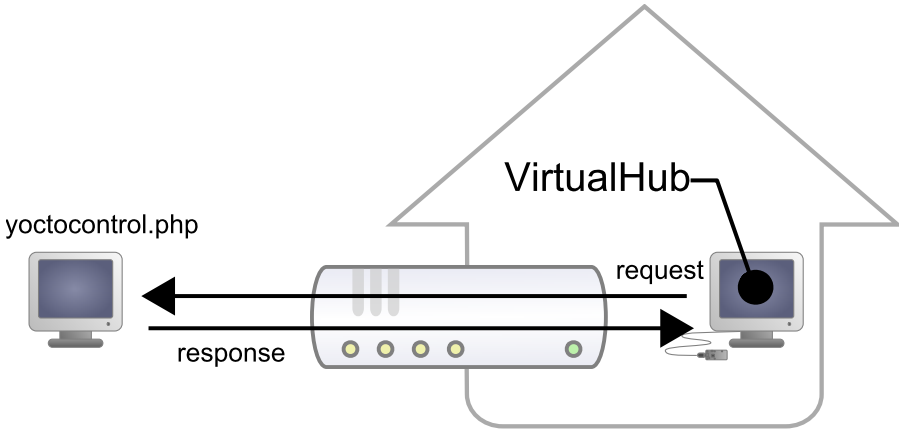

*Responses from request from LAN machines are routed.*




*But requests from the outside are blocked.*

Seeing Internet without being seen provides an enormous security advantage. However, this signifies that you cannot, a priori, set up your own web server at home to control a home automation installation from the outside. A solution to this problem, advised by numerous home automation system dealers, consists in providing outside visibility to your home automation server itself, by adding a routing rule in the NAT configuration of the DSL router. The issue of this solution is that it exposes the home automation server to external attacks.

The HTTP callback API solves this issue without having to modify the DSL router configuration. The module control script is located on an external site, and it is the *VirtualHub* which is in charge of calling it a regular intervals.



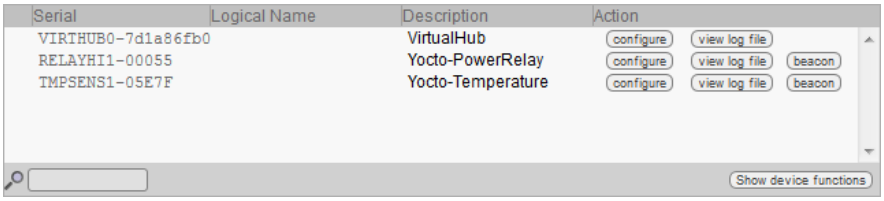

*The HTTP callback API uses the VirtualHub which initiates the requests.*

## Configuration

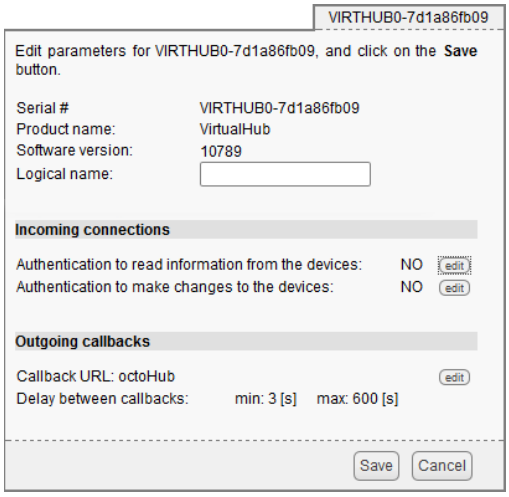
The callback API thus uses the *VirtualHub* as a gateway. All the communications are initiated by the *VirtualHub*. They are thus outgoing communications and therefore perfectly authorized by the DSL router.

You must configure the *VirtualHub* so that it calls the PHP script on a regular basis. To do so:

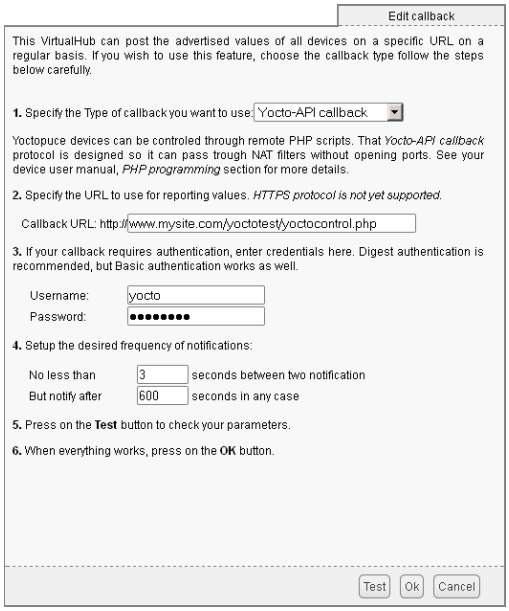
1. Launch a *VirtualHub*
2. Access its interface, usually 127.0.0.1:4444
3. Click on the **configure** button of the line corresponding to the *VirtualHub* itself
4. Click on the **edit** button of the **Outgoing callbacks** section




*Click on the "configure" button on the first line*




*Click on the "edit" button of the "Outgoing callbacks" section*




*And select "Yocto-API callback".*

You then only need to define the URL of the PHP script and, if need be, the user name and password to access this URL. Supported authentication methods are *basic* and *digest*. The second method is safer than the first one because it does not allow transfer of the password on the network.

### Usage

From the programmer standpoint, the only difference is at the level of the *yRegisterHub* function call. Instead of using an IP address, you must use the *callback* string (or *http://callback* which is equivalent).

```
include("yocto_api.php");
yRegisterHub("callback");
```

The remainder of the code stays strictly identical. On the *VirtualHub* interface, at the bottom of the configuration window for the HTTP callback API , there is a button allowing you to test the call to the PHP script.

Be aware that the PHP script controlling the modules remotely through the HTTP callback API can be called only by the *VirtualHub*. Indeed, it requires the information posted by the *VirtualHub* to function. To code a web site which controls Yoctopuce modules interactively, you must create a user interface which stores in a file or in a database the actions to be performed on the Yoctopuce modules. These actions are then read and run by the control script.

### Common issues

For the HTTP callback API to work, the PHP option *allow_url_fopen* must be set. Some web site hosts do not set it by default. The problem then manifests itself with the following error:

```
error: URL file-access is disabled in the server configuration
```

To set this option, you must create, in the repertory where the control PHP script is located, an *.htaccess* file containing the following line:

```
php_flag "allow_url_fopen" "On"
```

Depending on the security policies of the host, it is sometimes impossible to authorize this option at the root of the web site, or even to install PHP scripts receiving data from a POST HTTP. In this case, place the PHP script in a subdirectory.

### Limitations

This method that allows you to go through NAT filters cheaply has nevertheless a price. Communications being initiated by the *VirtualHub* at a more or less regular interval, reaction time to an event is clearly longer than if the Yoctopuce modules were driven directly. You can configure the reaction time in the specific window of the *VirtualHub*, but it is at least of a few seconds in the best case.

The *HTTP callback Yocto-API* mode is currently available in PHP, EcmaScript (Node.JS) and Java only.

## 8.5. Error handling

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

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

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

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

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

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

# 9. Using Yocto-GPS with C++

C++ is not the simplest language to master. However, if you take care to limit yourself to its essential functionalities, this language can very well be used for short programs quickly coded, and it has the advantage of being easily ported from one operating system to another. Under Windows, all the examples and the project models are tested with Microsoft Visual Studio 2010 Express, freely available on the Microsoft web site[1]. Under Mac OS X, all the examples and project models are tested with XCode 4, available on the App Store. Moreover, under Max OS X and under Linux, you can compile the examples using a command line with GCC using the provided `GNUmakefile`. In the same manner under Windows, a `Makefile` allows you to compile examples using a command line, fully knowing the compilation and linking arguments.

Yoctopuce C++ libraries[2] are integrally provided as source files. A section of the low-level library is written in pure C, but you should not need to interact directly with it: everything was done to ensure the simplest possible interaction from C++. The library is naturally also available as binary files, so that you can link it directly if you prefer.

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

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

## 9.1. Control of the Latitude function

A few lines of code are enough to use a Yocto-GPS. Here is the skeleton of a C++ code snipplet to use the Latitude function.

```
#include "yocto_api.h"
#include "yocto_latitude.h"

[...]
// Enable detection of USB devices
String  errmsg;
YAPI::RegisterHub("usb", errmsg);
[...]

// Retrieve the object used to interact with the device
```

[1] http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-cpp-express
[2] www.yoctopuce.com/EN/libraries.php

```
YLatitude *latitude;
latitude = YLatitude::FindLatitude("YGNSSMK1-123456.latitude");

// Hot-plug is easy: just check that the device is online
if(latitude->isOnline())
{
    // Use latitude->get_currentValue()
    [...]
}
```

Let's look at these lines in more details.

## yocto_api.h et yocto_latitude.h

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

## YAPI::RegisterHub

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

## YLatitude::FindLatitude

The `YLatitude::FindLatitude` function allows you to find a latitude sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-GPS module with serial number *YGNSSMK1-123456* which you have named "*MyModule*", and for which you have given the *latitude* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
YLatitude *latitude = YLatitude::FindLatitude("YGNSSMK1-123456.latitude");
YLatitude *latitude = YLatitude::FindLatitude("YGNSSMK1-123456.MyFunction");
YLatitude *latitude = YLatitude::FindLatitude("MyModule.latitude");
YLatitude *latitude = YLatitude::FindLatitude("MyModule.MyFunction");
YLatitude *latitude = YLatitude::FindLatitude("MyFunction");
```

`YLatitude::FindLatitude` returns an object which you can then use at will to control the latitude sensor.

## isOnline

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

## get_latitude

The `get_latitude()` method of the object returned by `yFindGps` provides the latitude currently measured by the Yocto-GPS. The value returned is a string, the format will vary according to the Yocto-GPS configuration. To get a floating point value, no matter the configuartion, use the YLatitude Class.

## A real example

Launch your C++ environment and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-GPS** of the Yoctopuce library. If you prefer to work with your favorite text editor, open the file `main.cpp`, and type `make` to build the example when you are done.

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

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

using namespace std;

static void usage(void)
{
  cout << "usage: demo <serial_number> " << endl;
  cout << "       demo <logical_name>" << endl;
  cout << "       demo any" << endl;
  u64 now = YAPI::GetTickCount();
  while (YAPI::GetTickCount() - now < 3000) {
    // wait 3 sec to show the message
  }
  exit(1);
}

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

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

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

  if (target == "any") {
    gps = YGps::FirstGps();
    if (gps == NULL) {
      cout << "No module connected (check USB cable)" << endl;
      return 1;
    }
  } else {
    gps = YGps::FindGps(target + ".gps");
  }

  while (1) {
    if (!gps->isOnline()) {
      cout << "Module not connected (check identification and USB cable)";
      break;
    }
    if (!gps->get_isFixed()) {
      cout << "Fixing.." << endl;
    } else {
      cout << gps->get_latitude() << "  " << gps->get_longitude()    << endl;
    }
    cout << "  (press Ctrl-C to exit)" << endl;
    YAPI::Sleep(1000, errmsg);
  }
  YAPI::FreeAPI();
  return 0;
}
```

## 9.2. Control of the module part

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

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

#include "yocto_api.h"

using namespace std;
```

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


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

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

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

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

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

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

## Changing the module settings

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

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

#include "yocto_api.h"

using namespace std;

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

```
}

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

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

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

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

  if (module->isOnline()) {
    if (argc >= 3) {
      string newname =  argv[2];
      if (!yCheckLogicalName(newname)) {
        cerr << "Invalid name (" << newname << ")" << endl;
        usage(argv[0]);
      }
      module->set_logicalName(newname);
      module->saveToFlash();
    }
    cout << "Current name: " << module->get_logicalName() << endl;
  } else {
    cout << argv[1] << " not connected (check identification and USB cable)"
         << endl;
  }
  YAPI::FreeAPI();
  return 0;
}
```

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

## Listing the modules

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

```
#include <iostream>

#include "yocto_api.h"

using namespace std;

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

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

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

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

```
  return 0;
}
```

## 9.3. Error handling

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

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

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

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

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

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

## 9.4. Integration variants for the C++ Yoctopuce library

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

### Integration in source format (recommended)

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

- It guaranties the respect of the compilation conventions of your project (32/64 bits, inclusion of debugging symbols, unicode or ASCII characters, etc.);

- It facilitates debugging if you are looking for the cause of a problem linked to the Yoctopuce library;
- It reduces the dependencies on third party components, for example in the case where you would need to recompile this project for another architecture in many years;
- It does not require the installation of a dynamic library specific to Yoctopuce on the final system, everything is in the executable.

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

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

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

## Integration as a static library

With the integration of the Yoctopuce library as a static library, you do not need to install a dynamic library specific to Yoctopuce, everything is in the executable.

To use the static library, you must first compile it using the shell script `build.sh` on UNIX, or `build.bat` on Windows. This script, located in the root directory of the library, detects the OS and recompiles all the corresponding libraries as well as the examples.

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

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

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

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

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

## Integration as a dynamic library

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

To use the dynamic library, you must first compile it using the shell script `build.sh` on UNIX, or `build.bat` on Windows. This script, located in the root directory of the library, detects the OS and recompiles all the corresponding libraries as well as the examples.

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

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

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

With GCC, the command line to compile is simply:

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

# 10. Using Yocto-GPS with Objective-C

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

Yoctopuce Objective-C libraries[1] are integrally provided as source files. A section of the low-level library is written in pure C, but you should not need to interact directly with it: everything was done to ensure the simplest possible interaction from Objective-C.

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

In order to keep them simple, all the examples provided in this documentation are console applications. Naturally, the libraries function in a strictly identical manner if you integrate them in an application with a graphical interface. You can find on Yoctopuce blog a detailed example[2] with video shots showing how to integrate the library into your projects.

## 10.1. Control of the Latitude function

A few lines of code are enough to use a Yocto-GPS. Here is the skeleton of a Objective-C code snipplet to use the Latitude function.

```
#import "yocto_api.h"
#import "yocto_latitude.h"

...
NSError *error;
[YAPI RegisterHub:@"usb": &error]
...
// On récupère l'objet représentant le module (ici connecté en local sur USB)
latitude = [YLatitude FindLatitude:@"YGNSSMK1-123456.latitude"];

// Pour gérer le hot-plug, on vérifie que le module est là
if([latitude isOnline])
{
```

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

```
    // Utiliser [latitude get_currentValue]
    ...
}
```

Let's look at these lines in more details.

## yocto_api.h and yocto_latitude.h

These two import files provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api.h` must always be used, `yocto_latitude.h` is necessary to manage modules containing a latitude sensor, such as Yocto-GPS.

## [YAPI RegisterHub]

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

## [Latitude FindLatitude]

The `[Latitude FindLatitude]` function allows you to find a latitude sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-GPS module with serial number *YGNSSMK1-123456* which you have named "*MyModule*", and for which you have given the *latitude* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
YLatitude *latitude = [Latitude FindLatitude:@"YGNSSMK1-123456.latitude"];
YLatitude *latitude = [Latitude FindLatitude:@"YGNSSMK1-123456.MyFunction"];
YLatitude *latitude = [Latitude FindLatitude:@"MyModule.latitude"];
YLatitude *latitude = [Latitude FindLatitude:@"MyModule.MyFunction"];
YLatitude *latitude = [Latitude FindLatitude:@"MyFunction"];
```

`[Latitude FindLatitude]` returns an object which you can then use at will to control the latitude sensor.

## isOnline

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

## get_latitude

The `get_latitude()` method of the object returned by `YGps.FindGps` provides the latitude currently measured by the Yocto-GPS. The value returned is a string, the format will vary according to the Yocto-GPS configuration. To get a floating point value, no matter the configuartion, use the YLatitude Class.

## A real example

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

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

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

static void usage(void)
{
  NSLog(@"usage: demo <serial_number> ");
  NSLog(@"       demo <logical_name>");
  NSLog(@"       demo any                 (use any discovered device)");
  exit(1);
```

```
}

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

  if (argc < 2) {
    usage();
  }

  @autoreleasepool {
    // Setup the API to use local USB devices
    if([YAPI RegisterHub:@"usb": &error] != YAPI_SUCCESS) {
      NSLog(@"RegisterHub error: %@", [error localizedDescription]);
      return 1;
    }
    NSString     *target = [NSString stringWithUTF8String:argv[1]];
    YGps *gps;
    if ([target isEqualToString:@"any"]) {
      gps = [YGps FirstGps];
      if (gps == NULL) {
        NSLog(@"No module connected (check USB cable)");
        return 1;
      }
    } else {
      gps = [YGps FindGps:[target stringByAppendingString:@".gps"]];
    }

    while(1) {
      if(![gps isOnline]) {
        NSLog(@"Module not connected (check identification and USB cable)\n");
        break;
      }
      if([gps get_isFixed] != Y_ISFIXED_TRUE) {
        NSLog(@"fixing");
      } else {
        NSLog(@"%@ %@ \n", [gps get_latitude], [gps get_longitude]);
      }

      NSLog(@"  (press Ctrl-C to exit)\n");
      [YAPI Sleep:1000:NULL];
    }
    [YAPI FreeAPI];
  }
  return 0;
}
```

## 10.2. Control of the module part

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

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

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


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

  @autoreleasepool {
    // Setup the API to use local USB devices
    if([YAPI RegisterHub:@"usb": &error] != YAPI_SUCCESS) {
      NSLog(@"RegisterHub error: %@", [error localizedDescription]);
      return 1;
    }
    if(argc < 2)
```

```
      usage(argv[0]);
    NSString *serial_or_name = [NSString stringWithUTF8String:argv[1]];
    // use serial or logical name
    YModule *module = [YModule FindModule:serial_or_name];
    if ([module isOnline]) {
      if (argc > 2) {
        if (strcmp(argv[2], "ON") == 0)
          [module setBeacon:Y_BEACON_ON];
        else
          [module setBeacon:Y_BEACON_OFF];
      }
      NSLog(@"serial:       %@\n", [module serialNumber]);
      NSLog(@"logical name: %@\n", [module logicalName]);
      NSLog(@"luminosity:   %d\n", [module luminosity]);
      NSLog(@"beacon:       ");
      if ([module beacon] == Y_BEACON_ON)
        NSLog(@"ON\n");
      else
        NSLog(@"OFF\n");
      NSLog(@"upTime:       %ld sec\n", [module upTime] / 1000);
      NSLog(@"USB current:  %d mA\n",  [module usbCurrent]);
      NSLog(@"logs:  %@\n",  [module get_lastLogs]);
    } else {
      NSLog(@"%@ not connected (check identification and USB cable)\n",
            serial_or_name);
    }
    [YAPI FreeAPI];
  }
  return 0;
}
```

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

## Changing the module settings

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

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

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


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

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

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

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

    if (module.isOnline) {
```

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

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

### Listing the modules

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

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

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

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

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

    YModule *module = [YModule FirstModule];
    while (module != nil) {
      NSLog(@"%@ %@", module.serialNumber, module.productName);
      module = [module nextModule];
    }
    [YAPI FreeAPI];
  }
  return 0;
}
```

## 10.3. Error handling

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

The simplest way to work around the problem is the one used in the short examples provided in this chapter: before accessing a module, check that it is online with the `isOnline` function, and then

hope that it will stay so during the fraction of a second necessary for the following code lines to run. This method is not perfect, but it can be sufficient in some cases. You must however be aware that you cannot completely exclude an error which would occur after the call to `isOnline` and which could crash the software. The only way to prevent this is to implement one of the two error handling techniques described below.

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

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

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

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

# 11. Using Yocto-GPS with Visual Basic .NET

VisualBasic has long been the most favored entrance path to the Microsoft world. Therefore, we had to provide our library for this language, even if the new trend is shifting to C#. All the examples and the project models are tested with Microsoft VisualBasic 2010 Express, freely available on the Microsoft web site[1].

## 11.1. Installation

Download the Visual Basic Yoctopuce library from the Yoctopuce web site[2]. There is no setup program, simply copy the content of the zip file into the directory of your choice. You mostly need the content of the `Sources` directory. The other directories contain the documentation and a few sample programs. All sample projects are Visual Basic 2010, projects, if you are using a previous version, you may have to recreate the projects structure from scratch.

## 11.2. Using the Yoctopuce API in a Visual Basic project

The Visual Basic.NET Yoctopuce library is composed of a DLL and of source files in Visual Basic. The DLL is not a .NET DLL, but a classic DLL, written in C, which manages the low level communications with the modules[3]. The source files in Visual Basic manage the high level part of the API. Therefore, your need both this DLL and the .vb files of the `sources` directory to create a project managing Yoctopuce modules.

### Configuring a Visual Basic project

The following indications are provided for Visual Studio Express 2010, but the process is similar for other versions. Start by creating your project. Then, on the *Solution Explorer* panel, right click on your project, and select "Add" and then "Add an existing item".

A file selection window opens. Select the `yocto_api.vb` file and the files corresponding to the functions of the Yoctopuce modules that your project is going to manage. If in doubt, select all the files.

You then have the choice between simply adding these files to your project, or to add them as links (the **Add** button is in fact a scroll-down menu). In the first case, Visual Studio copies the selected files into your project. In the second case, Visual Studio simply keeps a link on the original files. We recommend you to use links, which makes updates of the library much easier.

[1] http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-basic-express
[2] www.yoctopuce.com/EN/libraries.php
[3] The sources of this DLL are available in the C++ API

Then add in the same manner the `yapi.dll` DLL, located in the `Sources/dll` directory[4]. Then, from the **Solution Explorer** window, right click on the DLL, select **Properties** and in the **Properties** panel, set the **Copy to output folder** to **always**. You are now ready to use your Yoctopuce modules from Visual Studio.

In order to keep them simple, all the examples provided in this documentation are console applications. Naturally, the libraries function in a strictly identical manner if you integrate them in an application with a graphical interface.

# 11.3. Control of the Latitude function

A few lines of code are enough to use a Yocto-GPS. Here is the skeleton of a Visual Basic code snipplet to use the Latitude function.

```
[...]
' Enable detection of USB devices
Dim errmsg As String errmsg
YAPI.RegisterHub("usb", errmsg)
[...]

' Retrieve the object used to interact with the device
Dim latitude As YLatitude
latitude = YLatitude.FindLatitude("YGNSSMK1-123456.latitude")

' Hot-plug is easy: just check that the device is online
If (latitude.isOnline()) Then
   ' Use latitude.get_currentValue()
   [...]
End If

[...]
```

Let's look at these lines in more details.

## YAPI.RegisterHub

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

## YLatitude.FindLatitude

The `YLatitude.FindLatitude` function allows you to find a latitude sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-GPS module with serial number *YGNSSMK1-123456* which you have named "*MyModule*", and for which you have given the *latitude* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
latitude = YLatitude.FindLatitude("YGNSSMK1-123456.latitude")
latitude = YLatitude.FindLatitude("YGNSSMK1-123456.MyFunction")
latitude = YLatitude.FindLatitude("MyModule.latitude")
latitude = YLatitude.FindLatitude("MyModule.MyFunction")
latitude = YLatitude.FindLatitude("MyFunction")
```

`YLatitude.FindLatitude` returns an object which you can then use at will to control the latitude sensor.

## isOnline

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

[4] Remember to change the filter of the selection window, otherwise the DLL will not show.

## get_latitude

The `get_latitude()` method of the object returned by `yFindGps` provides the latitude currently measured by the Yocto-GPS. The value returned is a string, the format will vary according to the Yocto-GPS configuration. To get a floating point value, no matter the configuartion, use the YLatitude Class.

## A real example

Launch Microsoft VisualBasic and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-GPS** of the Yoctopuce library.

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

```
Module Module1

  Private Sub Usage()
    Dim execname = System.AppDomain.CurrentDomain.FriendlyName
    Console.WriteLine("Usage:")
    Console.WriteLine(execname + " <serial_number>")
    Console.WriteLine(execname + " <logical_name>")
    Console.WriteLine(execname + " any  ")
    System.Threading.Thread.Sleep(2500)

    End
  End Sub

  Sub Main()
    Dim argv() As String = System.Environment.GetCommandLineArgs()
    Dim errmsg As String = ""
    Dim target As String
    Dim gps As YGps

    If argv.Length < 2 Then Usage()

    target = argv(1)
    REM Setup the API to use local USB devices
    If (YAPI.RegisterHub("usb", errmsg) <> YAPI_SUCCESS) Then
      Console.WriteLine("RegisterHub error: " + errmsg)
      End
    End If

    If target = "any" Then
      gps = YGps.FirstGps()

      If gps Is Nothing Then
        Console.WriteLine("No module connected (check USB cable) ")
        End
      End If
    Else
      gps = YGps.FindGps(target + ".gps")
    End If

    While (True)
      If Not (gps.isOnline()) Then
        Console.WriteLine("Module not connected (check identification and USB cable)")
        End
      End If
      If (gps.get_isFixed() <> YGps.ISFIXED_TRUE) Then
        Console.WriteLine("Fixing...")
      Else
        Console.WriteLine(gps.get_latitude() + "  " + gps.get_longitude())
      End If
      Console.WriteLine("  (press Ctrl-C to exit)")
      YAPI.Sleep(1000, errmsg)
    End While
    YAPI.FreeAPI()
  End Sub

End Module
```

# 11.4. Control of the module part

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

```
Imports System.IO
Imports System.Environment

Module Module1

  Sub usage()
    Console.WriteLine("usage: demo <serial or logical name> [ON/OFF]")
    End
  End Sub

  Sub Main()
    Dim argv() As String = System.Environment.GetCommandLineArgs()
    Dim errmsg As String = ""
    Dim m As ymodule

    If (YAPI.RegisterHub("usb", errmsg) <> YAPI_SUCCESS) Then
      Console.WriteLine("RegisterHub error:" + errmsg)
      End
    End If

    If argv.Length < 2 Then usage()

    m = YModule.FindModule(argv(1)) REM use serial or logical name
    If (m.isOnline()) Then
      If argv.Length > 2 Then
        If argv(2) = "ON" Then m.set_beacon(Y_BEACON_ON)
        If argv(2) = "OFF" Then m.set_beacon(Y_BEACON_OFF)
      End If
      Console.WriteLine("serial:       " + m.get_serialNumber())
      Console.WriteLine("logical name: " + m.get_logicalName())
      Console.WriteLine("luminosity:   " + Str(m.get_luminosity()))
      Console.Write("beacon:       ")
      If (m.get_beacon() = Y_BEACON_ON) Then
        Console.WriteLine("ON")
      Else
        Console.WriteLine("OFF")
      End If
      Console.WriteLine("upTime:       " + Str(m.get_upTime() / 1000) + " sec")
      Console.WriteLine("USB current:  " + Str(m.get_usbCurrent()) + " mA")
      Console.WriteLine("Logs:")
      Console.WriteLine(m.get_lastLogs())
    Else
      Console.WriteLine(argv(1) + " not connected (check identification and USB cable)")
    End If
    YAPI.FreeAPI()
  End Sub

End Module
```

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

## Changing the module settings

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

```
Module Module1
```

```
  Sub usage()

    Console.WriteLine("usage: demo <serial or logical name> <new logical name>")
    End
  End Sub

  Sub Main()
    Dim argv() As String = System.Environment.GetCommandLineArgs()
    Dim errmsg As String = ""
    Dim newname As String
    Dim m As YModule

    If (argv.Length <> 3) Then usage()

    REM Setup the API to use local USB devices
    If YAPI.RegisterHub("usb", errmsg) <> YAPI_SUCCESS Then
      Console.WriteLine("RegisterHub error: " + errmsg)
      End
    End If

    m = YModule.FindModule(argv(1)) REM use serial or logical name
    If m.isOnline() Then
      newname = argv(2)
      If (Not YAPI.CheckLogicalName(newname)) Then
        Console.WriteLine("Invalid name (" + newname + ")")
        End
      End If
      m.set_logicalName(newname)
      m.saveToFlash() REM do not forget this
      Console.Write("Module: serial= " + m.get_serialNumber)
      Console.Write(" / name= " + m.get_logicalName())
    Else
      Console.Write("not connected (check identification and USB cable")
    End If
    YAPI.FreeAPI()

  End Sub

End Module
```

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

## Listing the modules

Obtaining the list of the connected modules is performed with the `yFirstModule()` function which returns the first module found. Then, you only need to call the `nextModule()` function of this object to find the following modules, and this as long as the returned value is not `Nothing`. Below a short example listing the connected modules.

```
Module Module1

  Sub Main()
    Dim M As ymodule
    Dim errmsg As String = ""

    REM Setup the API to use local USB devices
    If YAPI.RegisterHub("usb", errmsg) <> YAPI_SUCCESS Then
      Console.WriteLine("RegisterHub error: " + errmsg)
      End
    End If

    Console.WriteLine("Device list")
    M = YModule.FirstModule()
    While M IsNot Nothing
      Console.WriteLine(M.get_serialNumber() + " (" + M.get_productName() + ")")
      M = M.nextModule()
    End While
    YAPI.FreeAPI()
  End Sub
```

```
End Module
```

## 11.5. Error handling

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

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

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

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

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

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

# 12. Using Yocto-GPS with C#

C# (pronounced C-Sharp) is an object-oriented programming language promoted by Microsoft, it is somewhat similar to Java. Like Visual-Basic and Delphi, it allows you to create Windows applications quite easily. All the examples and the project models are tested with Microsoft C# 2010 Express, freely available on the Microsoft web site[1].

Our programming library is also compatible with *Mono*, the open source version of C# that also works on Linux and MacOS. You will find on our web site various articles that describe how to configure Mono to use our library.

## 12.1. Installation

Download the Visual C# Yoctopuce library from the Yoctopuce web site[2]. There is no setup program, simply copy the content of the zip file into the directory of your choice. You mostly need the content of the `Sources` directory. The other directories contain the documentation and a few sample programs. All sample projects are Visual C# 2010, projects, if you are using a previous version, you may have to recreate the projects structure from scratch.

## 12.2. Using the Yoctopuce API in a Visual C# project

The Visual C#.NET Yoctopuce library is composed of a DLL and of source files in Visual C#. The DLL is not a .NET DLL, but a classic DLL, written in C, which manages the low level communications with the modules[3]. The source files in Visual C# manage the high level part of the API. Therefore, your need both this DLL and the .cs files of the `sources` directory to create a project managing Yoctopuce modules.

### Configuring a Visual C# project

The following indications are provided for Visual Studio Express 2010, but the process is similar for other versions. Start by creating your project. Then, on the *Solution Explorer* panel, right click on your project, and select "Add" and then "Add an existing item".

A file selection window opens. Select the `yocto_api.cs` file and the files corresponding to the functions of the Yoctopuce modules that your project is going to manage. If in doubt, select all the files.

[1] http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-csharp-express
[2] www.yoctopuce.com/EN/libraries.php
[3] The sources of this DLL are available in the C++ API

You then have the choice between simply adding these files to your project, or to add them as links (the **Add** button is in fact a scroll-down menu). In the first case, Visual Studio copies the selected files into your project. In the second case, Visual Studio simply keeps a link on the original files. We recommend you to use links, which makes updates of the library much easier.

Then add in the same manner the `yapi.dll` DLL, located in the `Sources/dll` directory[4]. Then, from the **Solution Explorer** window, right click on the DLL, select **Properties** and in the **Properties** panel, set the **Copy to output folder** to **always**. You are now ready to use your Yoctopuce modules from Visual Studio.

In order to keep them simple, all the examples provided in this documentation are console applications. Naturally, the libraries function in a strictly identical manner if you integrate them in an application with a graphical interface.

## 12.3. Control of the Latitude function

A few lines of code are enough to use a Yocto-GPS. Here is the skeleton of a C# code snipplet to use the Latitude function.

```
[...]
// Enable detection of USB devices
string errmsg ="";
YAPI.RegisterHub("usb", errmsg);
[...]

// Retrieve the object used to interact with the device
YLatitude latitude = YLatitude.FindLatitude("YGNSSMK1-123456.latitude");

// Hot-plug is easy: just check that the device is online
if (latitude.isOnline())
{
    // Use latitude.get_currentValue()
    [...]
}
```

Let's look at these lines in more details.

### YAPI.RegisterHub

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

### YLatitude.FindLatitude

The `YLatitude.FindLatitude` function allows you to find a latitude sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-GPS module with serial number *YGNSSMK1-123456* which you have named "*MyModule*", and for which you have given the *latitude* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
latitude = YLatitude.FindLatitude("YGNSSMK1-123456.latitude");
latitude = YLatitude.FindLatitude("YGNSSMK1-123456.MyFunction");
latitude = YLatitude.FindLatitude("MyModule.latitude");
latitude = YLatitude.FindLatitude("MyModule.MyFunction");
latitude = YLatitude.FindLatitude("MyFunction");
```

`YLatitude.FindLatitude` returns an object which you can then use at will to control the latitude sensor.

[4] Remember to change the filter of the selection window, otherwise the DLL will not show.

## isOnline

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

## get_latitude

The `get_latitude()` method of the object returned by `YGps.FindGps` provides the latitude currently measured by the Yocto-GPS. The value returned is a string, the format will vary according to the Yocto-GPS configuration. To get a floating point value, no matter the configuartion, use the YLatitude Class.

## A real example

Launch Microsoft Visual C# and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-GPS** of the Yoctopuce library.

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

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
  class Program
  {
    static void usage()
    {
      string execname = System.AppDomain.CurrentDomain.FriendlyName;
      Console.WriteLine("Usage:");
      Console.WriteLine(execname + " <serial_number>");
      Console.WriteLine(execname + " <logical_name>");
      Console.WriteLine(execname + " any  ");
      System.Threading.Thread.Sleep(2500);
      Environment.Exit(0);
    }

    static void Main(string[] args)
    {
      string errmsg = "";
      string target;

      YGps gps;

      if (args.Length < 1) usage();
      target = args[0].ToUpper();

      // Setup the API to use local USB devices
      if (YAPI.RegisterHub("usb", ref errmsg) != YAPI.SUCCESS) {
        Console.WriteLine("RegisterHub error: " + errmsg);
        Environment.Exit(0);
      }

      if (target == "ANY") {
        gps = YGps.FirstGps();

        if (gps == null) {
          Console.WriteLine("No module connected (check USB cable) ");
          Environment.Exit(0);
        }
      } else {
        gps = YGps.FindGps(target + ".gps");
      }

      if (!gps.isOnline()) {
        Console.WriteLine("Module not connected");
        Console.WriteLine("check identification and USB cable");
        Environment.Exit(0);
      }
      while (gps.isOnline()) {
        if (gps.get_isFixed() != YGps.ISFIXED_TRUE)
```

```
          Console.WriteLine("fixing... ");
        else
          Console.WriteLine(gps.get_latitude() + "  " + gps.get_longitude());
        Console.WriteLine("  (press Ctrl-C to exit)");

        YAPI.Sleep(1000, ref errmsg);
      }
      YAPI.FreeAPI();
    }
  }
}
```

## 12.4. Control of the module part

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

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;


namespace ConsoleApplication1
{
  class Program
  {
    static void usage()
    {
      string execname = System.AppDomain.CurrentDomain.FriendlyName;
      Console.WriteLine("Usage:");
      Console.WriteLine(execname + " <serial or logical name> [ON/OFF]");
      System.Threading.Thread.Sleep(2500);
      Environment.Exit(0);
    }

    static void Main(string[] args)
    {
      YModule m;
      string errmsg = "";

      if (YAPI.RegisterHub("usb", ref errmsg) !=  YAPI.SUCCESS) {
        Console.WriteLine("RegisterHub error: " + errmsg);
        Environment.Exit(0);
      }


      if (args.Length < 1)  usage();

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

      if (m.isOnline()) {
        if (args.Length >= 2) {
          if (args[1].ToUpper() == "ON") {
            m.set_beacon(YModule.BEACON_ON);
          }
          if (args[1].ToUpper() == "OFF") {
            m.set_beacon(YModule.BEACON_OFF);
          }
        }

        Console.WriteLine("serial:       " + m.get_serialNumber());
        Console.WriteLine("logical name: " + m.get_logicalName());
        Console.WriteLine("luminosity:   " + m.get_luminosity().ToString());
        Console.Write("beacon:       ");
        if (m.get_beacon() == YModule.BEACON_ON)
          Console.WriteLine("ON");
        else
          Console.WriteLine("OFF");
        Console.WriteLine("upTime:       " + (m.get_upTime() / 1000 ).ToString() + " sec");
        Console.WriteLine("USB current:  " + m.get_usbCurrent().ToString() + " mA");
        Console.WriteLine("Logs:\r\n" + m.get_lastLogs());

      } else {
```

```
        Console.WriteLine(args[0] + " not connected (check identification and USB cable)");
      }
      YAPI.FreeAPI();
    }
  }
}
```

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

## Changing the module settings

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

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
  class Program
  {
    static void usage()
    {
      string execname = System.AppDomain.CurrentDomain.FriendlyName;
      Console.WriteLine("Usage:");
      Console.WriteLine("usage: demo <serial or logical name> <new logical name>");
      System.Threading.Thread.Sleep(2500);
      Environment.Exit(0);
    }

    static void Main(string[] args)
    {
      YModule m;
      string errmsg = "";
      string newname;

      if (args.Length != 2) usage();

      if (YAPI.RegisterHub("usb", ref errmsg) !=  YAPI.SUCCESS) {
        Console.WriteLine("RegisterHub error: " + errmsg);
        Environment.Exit(0);
      }

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

      if (m.isOnline()) {
        newname = args[1];
        if (!YAPI.CheckLogicalName(newname)) {
          Console.WriteLine("Invalid name (" + newname + ")");
          Environment.Exit(0);
        }

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

        Console.Write("Module: serial= " + m.get_serialNumber());
        Console.WriteLine(" / name= " + m.get_logicalName());
      } else {
        Console.Write("not connected (check identification and USB cable");
      }
      YAPI.FreeAPI();
    }
  }
}
```

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

### Listing the modules

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

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
  class Program
  {
    static void Main(string[] args)
    {
      YModule m;
      string errmsg = "";

      if (YAPI.RegisterHub("usb", ref errmsg) !=  YAPI.SUCCESS) {
        Console.WriteLine("RegisterHub error: " + errmsg);
        Environment.Exit(0);
      }

      Console.WriteLine("Device list");
      m = YModule.FirstModule();
      while (m != null) {
        Console.WriteLine(m.get_serialNumber() + " (" + m.get_productName() + ")");
        m = m.nextModule();
      }
      YAPI.FreeAPI();
    }
  }
}
```

## 12.5. Error handling

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

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

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

- If your code catches the exception and handles it, everything goes well.

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

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

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

# 13. Using the Yocto-GPS with Universal Windows Platform

Universal Windows Platform (UWP) is not a language per say, but a software platform created by Microsoft. This platform allows you to run a new type of applications: the universal Windows applications. These applications can work on all machines running under Windows 10. This includes computers, tablets, smart phones, XBox One, and also Windows IoT Core.

The Yoctopuce UWP library allows you to use Yoctopuce modules in a universal Windows application and is written in C# in its entirety. You can add it to a Visual Studio 2017[1] project.

## 13.1. Blocking and asynchronous functions

The Universal Windows Platform does not use the Win32 API but only the Windows Runtime API which is available on all the versions of Windows 10 and for any architecture. Thanks to this library, you can use UWP on all the Windows 10 versions, including Windows 10 IoT Core.

However, using the new UWP API has some consequences: the Windows Runtime API to access the USB ports is asynchronous, and therefore the Yoctopuce library must be asynchronous as well. Concretely, the asynchronous methods do not return a result directly but a `Task` or `Task<>` object and the result can be obtained later. Fortunately, the C# language, version 6, supports the `async` and `await` keywords, which simplifies using these functions enormously. You can thus use asynchronous functions in the same way as traditional functions as long as you respect the following two rules:

- The method is declared as asynchronous with the `async` keyword
- The `await` keyword is added when calling an asynchronous function

Example:

```
async Task<int> MyFunction(int  val)
{
    // do some long computation
    ...

    return result;
}

int res = await MyFunction(1234);
```

[1] https://www.visualstudio.com/vs/cordova/vs/

Our library follows these two rules and can therefore use the `await` notation.

For you not to have to wonder wether a function is asynchronous or not, there is the following convention: **all the public methods** of the UWP library **are asynchronous**, that is that you must call them with the `await` keyword, **except**:

- `GetTickCount()`, because measuring time in an asynchronous manner does not make a lot of sense...
- `FindModule()`, `FirstModule()`, `nextModule()`,... because detecting and enumerating modules is performed as a background task on internal structures which are managed transparently. It is therefore not necessary to use blocking functions while going though the lists of modules.

# 13.2. Installation

Download the Yoctopuce library for Universal Windows Platform from the Yoctopuce web site[2]. There is no installation software, simply copy the content of the zip file in a directory of your choice. You essentially need the content of the `Sources` directory. The other directories contain documentation and a few sample programs. Sample projects are Visual Studio 2017 projects. Visual Studio 2017 is available on the Microsoft web site[3].

# 13.3. Using the Yoctopuce API in a Visual Studio project

Start by creating your project. Then, from the **Solution Explorer** panel right click on your project and select **Add** then **Existing element** .

A file chooser opens: select all the files in the library `Sources` directory.

You then have the choice between simply adding the files to your project or adding them as a link (the **Add** button is actually a drop-down menu). In the first case, Visual Studio copies the selected files into your project. In the second case, Visual Studio simply creates a link to the original files. We recommend to use links, as a potential library update is thus much easier.

## The Package.appxmanifest file

By default a Universal Windows application doesn't have access rights to the USB ports. If you want to access USB devices, you must imperatively declare it in the `Package.appxmanifest` file.

Unfortunately, the edition window of this file doesn't allow this operation and you must modify the `Package.appxmanifest` file by hand. In the "Solution Explorer" panel, right click on the `Package.appxmanifest` and select "View Code".

In this XML file, we must add a `DeviceCapability` node in the `Capabilities` node. This node must have a "Name" attribute with a "humaninterfacedevice" value.

Inside this node, you must declare all the modules that can be used. Concretely, for each module, you must add a "Device" node with an "Id" attribute, which has for value a character string "vidpid:`USB_VENDORID` `USB_DEVICE_ID`". The Yoctopuce `USB_VENDORID` is 24e0 and you can find the `USB_DEVICE_ID` of each Yoctopuce device in the documentation in the "Characteristics" section. Finally, the "Device" node must contain a "Function" node with the "Type" attribute with a value of "usage:ff00 0001".

For the Yocto-GPS, here is what you must add in the "Capabilities" node:

```
<DeviceCapability Name="humaninterfacedevice">
    <!-- Yocto-GPS -->
    <Device Id="vidpid:24e0 0053">
      <Function Type="usage:ff00 0001" />
```

[2] www.yoctopuce.com/EN/libraries.php
[3] https://www.visualstudio.com/downloads/

```
      </Device>
    </DeviceCapability>
```

Unfortunately, it's not possible to write a rule authorizing all Yoctopuce modules. Therefore, you must imperatively add each module that you want to use.

# 13.4. Control of the Latitude function

A few lines of code are enough to use a Yocto-GPS. Here is the skeleton of a C# code snippet to use the Latitude function.

```
[...]
// Enable detection of USB devices
await YAPI.RegisterHub("usb");
[...]

// Retrieve the object used to interact with the device
YLatitude latitude = YLatitude.FindLatitude("YGNSSMK1-123456.latitude");

// Hot-plug is easy: just check that the device is online
if (await latitude.isOnline())
{
        // Use latitude.get_currentValue()
    [...]
}

[...]
```

Let us look at these lines in more details.

## YAPI.RegisterHub

The `YAPI.RegisterHub` function initializes the Yoctopuce API and indicates where the modules should be looked for. The parameter is the address of the virtual hub able to see the devices. If the string "usb" is passed as parameter, the API works with modules locally connected to the machine. If the initialization does not succeed, an exception is thrown.

## YLatitude.FindLatitude

The `YLatitude.FindLatitude` function allows you to find a latitude sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-GPS module with serial number *YGNSSMK1-123456* which you have named "*MyModule*", and for which you have given the *latitude* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
latitude = YLatitude.FindLatitude("YGNSSMK1-123456.latitude");
latitude = YLatitude.FindLatitude("YGNSSMK1-123456.MaFonction");
latitude = YLatitude.FindLatitude("MonModule.latitude");
latitude = YLatitude.FindLatitude("MonModule.MaFonction");
latitude = YLatitude.FindLatitude("MaFonction");
```

`YLatitude.FindLatitude` returns an object which you can then use at will to control the latitude sensor.

## isOnline

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

## get_latitude

The `get_latitude()` method of the object returned by `YGps.FindGps` provides the latitude currently measured by the Yocto-GPS. The value returned is a string, the format will vary according to the Yocto-GPS configuration. To get a floating point value, no matter the configuartion, use the YLatitude Class.

## 13.5. A real example

Launch Visual Studio and open the corresponding project provided in the directory **Examples/Doc-GettingStarted-Yocto-GPS** of the Yoctopuce library.

Visual Studio projects contain numerous files, and most of them are not linked to the use of the Yoctopuce library. To simplify reading the code, we regrouped all the code that uses the library in the `Demo` class, located in the `demo.cs` file. Properties of this class correspond to the different fields displayed on the screen, and the `Run()` method contains the code which is run when the "Start" button is pushed.

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

```
using System;
using System.Diagnostics;
using System.Threading.Tasks;
using Windows.UI.Xaml.Controls;
using com.yoctopuce.YoctoAPI;

namespace Demo
{
  public class Demo : DemoBase
  {
    public string HubURL { get; set; }
    public string Target { get; set; }

    public override async Task<int> Run()
    {
      try {
        await YAPI.RegisterHub(HubURL);

        YGps gps;

        if (Target.ToLower() == "any") {
          gps = YGps.FirstGps();
          if (gps == null) {
            WriteLine("No module connected (check USB cable) ");
            return -1;
          }
        } else {
          gps = YGps.FindGps(Target + ".gps");
        }

        while (await gps.isOnline()) {
          if (await gps.get_isFixed() != YGps.ISFIXED_TRUE)
            WriteLine("fixing... ");
          else
            WriteLine(await gps.get_latitude() + "  " + await gps.get_longitude());
          await YAPI.Sleep(1000);
        }

        WriteLine("Module not connected (check identification and USB cable)");
      } catch (YAPI_Exception ex) {
        WriteLine("error: " + ex.Message);
      }

      YAPI.FreeAPI();
      return 0;
    }
  }
}
```

## 13.6. Control of the module part

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

```
using System;
using System.Diagnostics;
```

```
using System.Threading.Tasks;
using Windows.UI.Xaml.Controls;
using com.yoctopuce.YoctoAPI;

namespace Demo
{
  public class Demo : DemoBase
  {

    public string HubURL { get; set; }
    public string Target { get; set; }
    public bool Beacon { get; set; }

    public override async Task<int> Run()
    {
      YModule m;
      string errmsg = "";

      if (await YAPI.RegisterHub(HubURL) != YAPI.SUCCESS) {
        WriteLine("RegisterHub error: " + errmsg);
        return -1;
      }
      m = YModule.FindModule(Target + ".module"); // use serial or logical name
      if (await m.isOnline()) {
        if (Beacon) {
          await m.set_beacon(YModule.BEACON_ON);
        } else {
          await m.set_beacon(YModule.BEACON_OFF);
        }

        WriteLine("serial: " + await m.get_serialNumber());
        WriteLine("logical name: " + await m.get_logicalName());
        WriteLine("luminosity: " + await m.get_luminosity());
        Write("beacon: ");
        if (await m.get_beacon() == YModule.BEACON_ON)
          WriteLine("ON");
        else
          WriteLine("OFF");
        WriteLine("upTime: " + (await m.get_upTime() / 1000) + " sec");
        WriteLine("USB current: " + await m.get_usbCurrent() + " mA");
        WriteLine("Logs:\r\n" + await m.get_lastLogs());
      } else {
        WriteLine(Target + " not connected  on" + HubURL +
                  "(check identification and USB cable)");
      }
      YAPI.FreeAPI();
      return 0;
    }
  }
}
```

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

## Changing the module settings

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

```
using System;
using System.Diagnostics;
using System.Threading.Tasks;
using Windows.UI.Xaml.Controls;
using com.yoctopuce.YoctoAPI;

namespace Demo
{
  public class Demo : DemoBase
  {
```

```
    public string HubURL { get; set; }
    public string Target { get; set; }
    public string LogicalName { get; set; }

    public override async Task<int> Run()
    {
      try {
        YModule m;

        await YAPI.RegisterHub(HubURL);

        m = YModule.FindModule(Target); // use serial or logical name
        if (await m.isOnline()) {
          if (!YAPI.CheckLogicalName(LogicalName)) {
            WriteLine("Invalid name (" + LogicalName + ")");
            return -1;
          }

          await m.set_logicalName(LogicalName);
          await m.saveToFlash(); // do not forget this
          Write("Module: serial= " + await m.get_serialNumber());
          WriteLine(" / name= " + await m.get_logicalName());
        } else {
          Write("not connected (check identification and USB cable");
        }
      } catch (YAPI_Exception ex) {
        WriteLine("RegisterHub error: " + ex.Message);
      }
      YAPI.FreeAPI();
      return 0;
    }
  }
}
```

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

## Listing the modules

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

```
using System;
using System.Diagnostics;
using System.Threading.Tasks;
using Windows.UI.Xaml.Controls;
using com.yoctopuce.YoctoAPI;

namespace Demo
{
  public class Demo : DemoBase
  {
    public string HubURL { get; set; }

    public override async Task<int> Run()
    {
      YModule m;
      try {
        await YAPI.RegisterHub(HubURL);

        WriteLine("Device list");
        m = YModule.FirstModule();
        while (m != null) {
          WriteLine(await m.get_serialNumber()
                    + " (" + await m.get_productName() + ")");
          m = m.nextModule();
        }
      } catch (YAPI_Exception ex) {
        WriteLine("Error:" + ex.Message);
```

```
        }
        YAPI.FreeAPI();
        return 0;
      }
    }
}
```

## 13.7. Error handling

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

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

In the Universal Windows Platform library, error handling is implemented with exceptions. You must therefore intercept and correctly handle these exceptions if you want to have a reliable project which does not crash as soon as you disconnect a module.

Library thrown exceptions are always of the YAPI_Exception type, so you can easily separate them from other exceptions in a `try{...} catch{...}` block.

Example:

```
try {
        ....
} catch (YAPI_Exception ex) {
        Debug.WriteLine("Exception from Yoctopuce lib:" + ex.Message);
} catch (Exception ex) {
        Debug.WriteLine("Other exceptions :" + ex.Message);
}
```

# 14. Using Yocto-GPS with Delphi

Delphi is a descendent of Turbo-Pascal. Originally, Delphi was produced by Borland, Embarcadero now edits it. The strength of this language resides in its ease of use, as anyone with some notions of the Pascal language can develop a Windows application in next to no time. Its only disadvantage is to cost something[1].

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

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

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

## 14.1. Preparation

Go to the Yoctopuce web site and download the Yoctopuce Delphi libraries[3]. Uncompress everything in a directory of your choice, add the subdirectory *sources* in the list of directories of Delphi libraries.[4]

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

## 14.2. Control of the Latitude function

A few lines of code are enough to use a Yocto-GPS. Here is the skeleton of a Delphi code snipplet to use the Latitude function.

```
uses yocto_api, yocto_latitude;

var errmsg: string;
    latitude: TYLatitude;

[...]
```

[1] Actually, Borland provided free versions (for personal use) of Delphi 2006 and 2007. Look for them on the Internet, you may still be able to download them.
[2] Delphi libraries are regularly tested with Delphi 5 and Delphi XE2.
[3] www.yoctopuce.com/EN/libraries.php
[4] Use the **Tools / Environment options** menu.

```
// Enable detection of USB devices
yRegisterHub('usb',errmsg)
[...]

// Retrieve the object used to interact with the device
latitude = yFindLatitude("YGNSSMK1-123456.latitude")

// Hot-plug is easy: just check that the device is online
if latitude.isOnline() then
    begin
        // Use latitude.get_currentValue()
        [...]
    end;
[...]
```

Let's look at these lines in more details.

## yocto_api and yocto_latitude

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

## yRegisterHub

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

## yFindLatitude

The `yFindLatitude` function allows you to find a latitude sensor from the serial number of the module on which it resides and from its function name. You can also use logical names, as long as you have initialized them. Let us imagine a Yocto-GPS module with serial number *YGNSSMK1-123456* which you have named "*MyModule*", and for which you have given the *latitude* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
latitude := yFindLatitude("YGNSSMK1-123456.latitude");
latitude := yFindLatitude("YGNSSMK1-123456.MyFunction");
latitude := yFindLatitude("MyModule.latitude");
latitude := yFindLatitude("MyModule.MyFunction");
latitude := yFindLatitude("MyFunction");
```

`yFindLatitude` returns an object which you can then use at will to control the latitude sensor.

## isOnline

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

## get_latitude

The `get_latitude()` method of the object returned by `yFindGps` provides the latitude currently measured by the Yocto-GPS. The value returned is a string, the format will vary according to the Yocto-GPS configuration. To get a floating point value, no matter the configuartion, use the YLatitude Class.

## A real example

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

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

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

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

var
  gps : TYGps;
  errmsg : string;
  done   : boolean;

begin

  if (paramcount<1) then usage();

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

  if paramstr(1)='any' then
    begin
      // try to find  the first temperature gps available
      gps := yFirstGps();
      if gps=nil then
         begin
           writeln('No module connected (check USB cable)');
           halt;
         end
       end
   else  // or use the one specified on the commande line
    gps:= YFindGps(paramstr(1)+'.gps');

  // let's poll
  done := false;
  repeat
    if (gps.isOnline()) then
     begin
       if (gps.get_isFixed()<>Y_ISFIXED_TRUE) then
         Writeln('fixing')
       else
         writeln(gps.get_latitude()+'  '+gps.get_longitude());

       Writeln('   (press Ctrl-C to exit)');
       Sleep(1000);
     end
    else
     begin
       Writeln('Module not connected (check identification and USB cable)');
       done := true;
     end;
  until done;
  yFreeAPI();

end.
```

## 14.3. Control of the module part

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

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

const
  serial = 'YGNSSMK1-123456'; // use serial number or logical name

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


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

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

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

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

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

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

## Changing the module settings

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

forget its current settings by using the `revertFromFlash()` method. The short example below allows you to modify the logical name of a module.

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

const
  serial = 'YGNSSMK1-123456'; // use serial number or logical name

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

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

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

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

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

## Listing the modules

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

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

var
  module : TYModule;
  errmsg : string;

begin
  // Setup the API to use local USB devices
  if yRegisterHub('usb', errmsg)<>YAPI_SUCCESS then
  begin
```

```
    Write('RegisterHub error: '+errmsg);
    exit;
  end;

  Writeln('Device list');

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

end.
```

## 14.4. Error handling

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

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

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

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

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

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

# 15. Using the Yocto-GPS with Python

Python is an interpreted object oriented language developed by Guido van Rossum. Among its advantages is the fact that it is free, and the fact that it is available for most platforms, Windows as well as UNIX. It is an ideal language to write small scripts on a napkin. The Yoctopuce library is compatible with Python 2.6+ and 3+. It works under Windows, Mac OS X, and Linux, Intel as well as ARM. The library was tested with Python 2.6 and Python 3.2. Python interpreters are available on the Python web site[1].

## 15.1. Source files

The Yoctopuce library classes[2] for Python that you will use are provided as source files. Copy all the content of the *Sources* directory in the directory of your choice and add this directory to the *PYTHONPATH* environment variable. If you use an IDE to program in Python, refer to its documentation to configure it so that it automatically finds the API source files.

## 15.2. Dynamic library

A section of the low-level library is written in C, but you should not need to interact directly with it: it is provided as a DLL under Windows, as a *.so* files under UNIX, and as a *.dylib* file under Mac OS X. Everything was done to ensure the simplest possible interaction from Python: the distinct versions of the dynamic library corresponding to the distinct operating systems and architectures are stored in the *cdll* directory. The API automatically loads the correct file during its initialization. You should not have to worry about it.

If you ever need to recompile the dynamic library, its complete source code is located in the Yoctopuce C++ library.

In order to keep them simple, all the examples provided in this documentation are console applications. Naturally, the libraries function in a strictly identical manner if you integrate them in an application with a graphical interface.

## 15.3. Control of the Latitude function

A few lines of code are enough to use a Yocto-GPS. Here is the skeleton of a Python code snipplet to use the Latitude function.

[1] http://www.python.org/download/
[2] www.yoctopuce.com/EN/libraries.php

```
[...]
# Enable detection of USB devices
errmsg=YRefParam()
YAPI.RegisterHub("usb",errmsg)
[...]

# Retrieve the object used to interact with the device
latitude = YLatitude.FindLatitude("YGNSSMK1-123456.latitude")

# Hot-plug is easy: just check that the device is online
if latitude.isOnline():
    # Use latitude.get_currentValue()
    [...]

[...]
```

Let's look at these lines in more details.

## YAPI.RegisterHub

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

## YLatitude.FindLatitude

The `YLatitude.FindLatitude` function allows you to find a latitude sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-GPS module with serial number *YGNSSMK1-123456* which you have named "*MyModule*", and for which you have given the *latitude* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
latitude = YLatitude.FindLatitude("YGNSSMK1-123456.latitude")
latitude = YLatitude.FindLatitude("YGNSSMK1-123456.MyFunction")
latitude = YLatitude.FindLatitude("MyModule.latitude")
latitude = YLatitude.FindLatitude("MyModule.MyFunction")
latitude = YLatitude.FindLatitude("MyFunction")
```

`YLatitude.FindLatitude` returns an object which you can then use at will to control the latitude sensor.

## isOnline

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

## get_latitude

The `get_latitude()` method of the object returned by `YGps.FindGps` provides the latitude currently measured by the Yocto-GPS. The value returned is a string, the format will vary according to the Yocto-GPS configuration. To get a floating point value, no matter the configuartion, use the YLatitude Class.

## A real example

Launch Python and open the corresponding sample script provided in the directory **Examples/Doc-GettingStarted-Yocto-GPS** of the Yoctopuce library.

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

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import os, sys
```

```
from yocto_api import *
from yocto_gps import *


def usage():
    scriptname = os.path.basename(sys.argv[0])
    print("Usage:")
    print(scriptname + ' <serial_number>')
    print(scriptname + ' <logical_name>')
    print(scriptname + ' any  ')
    sys.exit()


def die(msg):
    sys.exit(msg + ' (check USB cable)')


if len(sys.argv) < 2:
    usage()
target = sys.argv[1]

# Setup the API to use local USB devices
errmsg = YRefParam()
if YAPI.RegisterHub("usb", errmsg) != YAPI.SUCCESS:
    sys.exit("init error" + errmsg.value)

if target == 'any':
    # retreive any gps
    gps = YGps.FirstGps()
    if gps is None:
        die('No module connected')
else:
    gps = YGps.FindGps(target + '.gps')

if not (gps.isOnline()):
    die('device not connected')

while gps.isOnline():
    if gps.get_isFixed() != YGps.ISFIXED_TRUE:
        print("Fixing...")
    else:
        print(gps.get_latitude() + "  " + gps.get_longitude())
    YAPI.Sleep(1000)
YAPI.FreeAPI()
```

## 15.4. Control of the module part

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

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import os, sys

from yocto_api import *


def usage():
    sys.exit("usage: demo <serial or logical name> [ON/OFF]")


errmsg = YRefParam()
if YAPI.RegisterHub("usb", errmsg) != YAPI.SUCCESS:
    sys.exit("RegisterHub error: " + str(errmsg))

if len(sys.argv) < 2:
    usage()

m = YModule.FindModule(sys.argv[1])  # # use serial or logical name

if m.isOnline():
    if len(sys.argv) > 2:
        if sys.argv[2].upper() == "ON":
```

```
            m.set_beacon(YModule.BEACON_ON)
        if sys.argv[2].upper() == "OFF":
            m.set_beacon(YModule.BEACON_OFF)

    print("serial:       " + m.get_serialNumber())
    print("logical name: " + m.get_logicalName())
    print("luminosity:   " + str(m.get_luminosity()))
    if m.get_beacon() == YModule.BEACON_ON:
        print("beacon:       ON")
    else:
        print("beacon:       OFF")
    print("upTime:       " + str(m.get_upTime() / 1000) + " sec")
    print("USB current:  " + str(m.get_usbCurrent()) + " mA")
    print("logs:\n" + m.get_lastLogs())
else:
    print(sys.argv[1] + " not connected (check identification and USB cable)")
YAPI.FreeAPI()
```

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

## Changing the module settings

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

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import os, sys

from yocto_api import *


def usage():
    sys.exit("usage: demo <serial or logical name> <new logical name>")


if len(sys.argv) != 3:
    usage()

errmsg = YRefParam()
if YAPI.RegisterHub("usb", errmsg) != YAPI.SUCCESS:
    sys.exit("RegisterHub error: " + str(errmsg))

m = YModule.FindModule(sys.argv[1])  # use serial or logical name
if m.isOnline():
    newname = sys.argv[2]
    if not YAPI.CheckLogicalName(newname):
        sys.exit("Invalid name (" + newname + ")")
    m.set_logicalName(newname)
    m.saveToFlash()  # do not forget this
    print("Module: serial= " + m.get_serialNumber() + " / name= " + m.get_logicalName())
else:
    sys.exit("not connected (check identification and USB cable")
YAPI.FreeAPI()
```

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

### Listing the modules

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

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import os, sys


from yocto_api import *

errmsg = YRefParam()

# Setup the API to use local USB devices
if YAPI.RegisterHub("usb", errmsg) != YAPI.SUCCESS:
    sys.exit("init error" + str(errmsg))

print('Device list')

module = YModule.FirstModule()
while module is not None:
    print(module.get_serialNumber() + ' (' + module.get_productName() + ')')
    module = module.nextModule()
YAPI.FreeAPI()
```

## 15.5. Error handling

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

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

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

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

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

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

# 16. Using the Yocto-GPS with Java

Java is an object oriented language created by Sun Microsystem. Beside being free, its main strength is its portability. Unfortunately, this portability has an excruciating price. In Java, hardware abstraction is so high that it is almost impossible to work directly with the hardware. Therefore, the Yoctopuce API does not support native mode in regular Java. The Java API needs a Virtual Hub to communicate with Yoctopuce devices.

## 16.1. Getting ready

Go to the Yoctopuce web site and download the following items:

- The Java programming library[1]
- The VirtualHub software[2] for Windows, Mac OS X or Linux, depending on your OS

The library is available as source files as well as a *jar* file. Decompress the library files in a folder of your choice, connect your modules, run the VirtualHub software, and you are ready to start your first tests. You do not need to install any driver.

In order to keep them simple, all the examples provided in this documentation are console applications. Naturally, the libraries function in a strictly identical manner if you integrate them in an application with a graphical interface.

## 16.2. Control of the Latitude function

A few lines of code are enough to use a Yocto-GPS. Here is the skeleton of a Java code snippet to use the Latitude function.

```
[...]
// Get access to your device, through the VirtualHub running locally
YAPI.RegisterHub("127.0.0.1");
[...]

// Retrieve the object used to interact with the device
latitude = YLatitude.FindLatitude("YGNSSMK1-123456.latitude");

// Hot-plug is easy: just check that the device is online
if (latitude.isOnline())
{
```

[1] www.yoctopuce.com/EN/libraries.php
[2] www.yoctopuce.com/EN/virtualhub.php

```
    // Use latitude.get_currentValue()
    [...]
}

[...]
```

Let us look at these lines in more details.

## YAPI.RegisterHub

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

## YLatitude.FindLatitude

The `YLatitude.FindLatitude` function allows you to find a latitude sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-GPS module with serial number *YGNSSMK1-123456* which you have named "*MyModule*", and for which you have given the *latitude* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
latitude = YLatitude.FindLatitude("YGNSSMK1-123456.latitude")
latitude = YLatitude.FindLatitude("YGNSSMK1-123456.MyFunction")
latitude = YLatitude.FindLatitude("MyModule.latitude")
latitude = YLatitude.FindLatitude("MyModule.MyFunction")
latitude = YLatitude.FindLatitude("MyFunction")
```

`YLatitude.FindLatitude` returns an object which you can then use at will to control the latitude sensor.

## isOnline

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

## get_latitude

The `get_latitude()` method of the object returned by `YGps.FindGps` provides the latitude currently measured by the Yocto-GPS. The value returned is a string, the format will vary according to the Yocto-GPS configuration. To get a floating point value, no matter the configuartion, use the YLatitude Class.

## A real example

Launch you Java environment and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-GPS** of the Yoctopuce library.

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

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

public class Demo {

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

```
        if (args.length == 0) {
            gps = YGps.FirstGps();
            if (gps == null) {
                System.out.println("No module connected (check USB cable)");
                System.exit(1);
            }
        } else {
            gps = YGps.FindGps(args[0] + ".gps");
        }

        while (true) {
            try {
                if (gps.get_isFixed() != YGps.ISFIXED_TRUE) {
                    System.out.println("fixing...");
                } else {
                    System.out.println(gps.get_latitude() + "  " + gps.get_longitude());
                }
                System.out.println("  (press Ctrl-C to exit)");
                YAPI.Sleep(1000);
            } catch (YAPI_Exception ex) {
                System.out.println("Module not connected (check identification and USB
cable)");
                break;
            }
        }

        YAPI.FreeAPI();
    }
}
```

## 16.3. Control of the module part

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

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

public class Demo {

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

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

        try {
            if (args.length > 1) {
                if (args[1].equalsIgnoreCase("ON")) {
                    module.setBeacon(YModule.BEACON_ON);
                } else {
                    module.setBeacon(YModule.BEACON_OFF);
```

```
                }
            }
            System.out.println("serial:       " + module.get_serialNumber());
            System.out.println("logical name: " + module.get_logicalName());
            System.out.println("luminosity:   " + module.get_luminosity());
            if (module.get_beacon() == YModule.BEACON_ON) {
                System.out.println("beacon:       ON");
            } else {
                System.out.println("beacon:       OFF");
            }
            System.out.println("upTime:       " + module.get_upTime() / 1000 + " sec");
            System.out.println("USB current:  " + module.get_usbCurrent() + " mA");
            System.out.println("logs:\n" + module.get_lastLogs());
        } catch (YAPI_Exception ex) {
            System.out.println(args[1] + " not connected (check identification and USB
cable)");
        }
        YAPI.FreeAPI();
    }
}
```

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

## Changing the module settings

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

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

public class Demo {

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

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

        YModule m;
        String newname;

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

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

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

            System.out.println("Module: serial= " + m.get_serialNumber());
            System.out.println(" / name= " + m.get_logicalName());
```

```
        } catch (YAPI_Exception ex) {
            System.out.println("Module " + args[0] + "not connected (check identification
and USB cable)");
            System.out.println(ex.getMessage());
            System.exit(1);
        }

        YAPI.FreeAPI();
    }
}
```

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

### Listing the modules

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

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

public class Demo {

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

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

## 16.4. Error handling

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

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

you cannot completely exclude an error which would occur after the call to `isOnline` and which could crash the software.

In the Java API, error handling is implemented with exceptions. Therefore you must catch and handle correctly all exceptions that might be thrown by the API if you do not want your software to crash as soon as you unplug a device.

# 17. Using the Yocto-GPS with Android

To tell the truth, Android is not a programming language, it is an operating system developed by Google for mobile appliances such as smart phones and tablets. But it so happens that under Android everything is programmed with the same programming language: Java. Nevertheless, the programming paradigms and the possibilities to access the hardware are slightly different from classical Java, and this justifies a separate chapter on Android programming.

## 17.1. Native access and VirtualHub

In the opposite to the classical Java API, the Java for Android API can access USB modules natively. However, as there is no VirtualHub running under Android, it is not possible to remotely control Yoctopuce modules connected to a machine under Android. Naturally, the Java for Android API remains perfectly able to connect itself to a VirtualHub running on another OS.

## 17.2. Getting ready

Go to the Yoctopuce web site and download the Java for Android programming library[1]. The library is available as source files, and also as a jar file. Connect your modules, decompress the library files in the directory of your choice, and configure your Android programming environment so that it can find them.

To keep them simple, all the examples provided in this documentation are snippets of Android applications. You must integrate them in your own Android applications to make them work. However, your can find complete applications in the examples provided with the Java for Android library.

## 17.3. Compatibility

In an ideal world, you would only need to have a smart phone running under Android to be able to make Yoctopuce modules work. Unfortunately, it is not quite so in the real world. A machine running under Android must fulfil to a few requirements to be able to manage Yoctopuce USB modules natively.

[1] www.yoctopuce.com/EN/libraries.php

### Android 4.x

Android 4.0 (api 14) and following are officially supported. Theoretically, support of USB *host* functions since Android 3.1. But be aware that the Yoctopuce Java for Android API is regularly tested only from Android 4 onwards.

### USB *host* support

Naturally, not only must your machine have a USB port, this port must also be able to run in *host* mode. In *host* mode, the machine literally takes control of the devices which are connected to it. The USB ports of a desktop computer, for example, work in *host* mode. The opposite of the *host* mode is the *device* mode. USB keys, for instance, work in *device* mode: they must be controlled by a *host*. Some USB ports are able to work in both modes, they are *OTG* (*On The Go*) ports. It so happens that many mobile devices can only work in *device* mode: they are designed to be connected to a charger or a desktop computer, and nothing else. It is therefore highly recommended to pay careful attention to the technical specifications of a product working under Android before hoping to make Yoctopuce modules work with it.

Unfortunately, having a correct version of Android and USB ports working in *host* mode is not enough to guaranty that Yoctopuce modules will work well under Android. Indeed, some manufacturers configure their Android image so that devices other than keyboard and mass storage are ignored, and this configuration is hard to detect. As things currently stand, the best way to know if a given Android machine works with Yoctopuce modules consists in trying.

### Supported hardware

The library is tested and validated on the following machines:

- Samsung Galaxy S3
- Samsung Galaxy Note 2
- Google Nexus 5
- Google Nexus 7
- Acer Iconia Tab A200
- Asus Tranformer Pad TF300T
- Kurio 7

If your Android machine is not able to control Yoctopuce modules natively, you still have the possibility to remotely control modules driven by a VirtualHub on another OS, or a YoctoHub [2].

## 17.4. Activating the USB port under Android

By default, Android does not allow an application to access the devices connected to the USB port. To enable your application to interact with a Yoctopuce module directly connected on your tablet on a USB port, a few additional steps are required. If you intend to interact only with modules connected on another machine through the network, you can ignore this section.

In your `AndroidManifest.xml`, you must declare using the "USB Host" functionality by adding the `<uses-feature android:name="android.hardware.usb.host" />` tag in the `manifest` section.

```
<manifest ...>
    ...
    <uses-feature android:name="android.hardware.usb.host" />;
    ...
</manifest>
```

When first accessing a Yoctopuce module, Android opens a window to inform the user that the application is going to access the connected module. The user can deny or authorize access to the device. If the user authorizes the access, the application can access the connected device as long as

[2] Yoctohubs are a plug and play way to add network connectivity to your Yoctopuce devices. more info on http://www.yoctopuce.com/EN/products/category/extensions-and-networking

it stays connected. To enable the Yoctopuce library to correctly manage these authorizations, your must provide a pointer on the application context by calling the EnableUSBHost method of the YAPI class before the first USB access. This function takes as arguments an object of the `android.content.Context` class (or of a subclass). As the Activity class is a subclass of `Context`, it is simpler to call `YAPI.EnableUSBHost(this);` in the method `onCreate` of your application. If the object passed as parameter is not of the correct type, a `YAPI_Exception` exception is generated.

```
...
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    try {
                // Pass the application Context to the Yoctopuce Library
        YAPI.EnableUSBHost(this);
        } catch (YAPI_Exception e) {
                Log.e("Yocto",e.getLocalizedMessage());
        }
}
...
```

### Autorun

It is possible to register your application as a default application for a USB module. In this case, as soon as a module is connected to the system, the application is automatically launched. You must add *<action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"/>* in the section *<intent-filter>* of the main activity. The section *<activity>* must have a pointer to an XML file containing the list of USB modules which can run the application.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    ...
    <uses-feature android:name="android.hardware.usb.host" />
    ...
    <application ... >
        <activity
            android:name=".MainActivity" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>

            <meta-data
                android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"
                android:resource="@xml/device_filter" />
        </activity>
    </application>

</manifest>
```

The XML file containing the list of modules allowed to run the application must be saved in the res/xml directory. This file contains a list of USB *vendorId* and *deviceID* in decimal. The following example runs the application as soon as a Yocto-Relay or a YoctoPowerRelay is connected. You can find the vendorID and the deviceID of Yoctopuce modules in the characteristics section of the documentation.

```
<?xml version="1.0" encoding="utf-8"?>

<resources>
    <usb-device vendor-id="9440" product-id="12" />
    <usb-device vendor-id="9440" product-id="13" />
</resources>
```

## 17.5. Control of the Latitude function

A few lines of code are enough to use a Yocto-GPS. Here is the skeleton of a Java code snippet to use the Latitude function.

```
[...]
// Enable detection of USB devices
YAPI.EnableUSBHost(this);
YAPI.RegisterHub("usb");
[...]
// Retrieve the object used to interact with the device
latitude = YLatitude.FindLatitude("YGNSSMK1-123456.latitude");

// Hot-plug is easy: just check that the device is online
if (latitude.isOnline()) {
    // Use latitude.get_currentValue()
    [...]
}

[...]
```

Let us look at these lines in more details.

## YAPI.EnableUSBHost

The `YAPI.EnableUSBHost` function initializes the API with the Context of the current application. This function takes as argument an object of the `android.content.Context` class (or of a subclass). If you intend to connect your application only to other machines through the network, this function is facultative.

## YAPI.RegisterHub

The `yAPI.RegisterHub` function initializes the Yoctopuce API and indicates where the modules should be looked for. The parameter is the address of the virtual hub able to see the devices. If the string "usb" is passed as parameter, the API works with modules locally connected to the machine. If the initialization does not succeed, an exception is thrown.

## YLatitude.FindLatitude

The `YLatitude.FindLatitude` function allows you to find a latitude sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-GPS module with serial number *YGNSSMK1-123456* which you have named "*MyModule*", and for which you have given the *latitude* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
latitude = YLatitude.FindLatitude("YGNSSMK1-123456.latitude")
latitude = YLatitude.FindLatitude("YGNSSMK1-123456.MyFunction")
latitude = YLatitude.FindLatitude("MyModule.latitude")
latitude = YLatitude.FindLatitude("MyModule.MyFunction")
latitude = YLatitude.FindLatitude("MyFunction")
```

`YLatitude.FindLatitude` returns an object which you can then use at will to control the latitude sensor.

## isOnline

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

## get_latitude

The `get_latitude()` method of the object returned by `YGps.FindGps` provides the latitude currently measured by the Yocto-GPS. The value returned is a string, the format will vary according to the Yocto-GPS configuration. To get a floating point value, no matter the configuartion, use the YLatitude Class.

## A real example

Launch you Java environment and open the corresponding sample project provided in the directory **Examples//Doc-Examples** of the Yoctopuce library.

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

```
package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.os.Handler;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemSelectedListener;
import android.widget.ArrayAdapter;
import android.widget.Spinner;
import android.widget.TextView;

import com.yoctopuce.YoctoAPI.YAPI;
import com.yoctopuce.YoctoAPI.YAPI_Exception;
import com.yoctopuce.YoctoAPI.YAltitude;
import com.yoctopuce.YoctoAPI.YGps;
import com.yoctopuce.YoctoAPI.YModule;
import com.yoctopuce.YoctoAPI.YPressure;
import com.yoctopuce.YoctoAPI.YTemperature;

public class GettingStarted_Yocto_GPS extends Activity implements OnItemSelectedListener {

    private ArrayAdapter<String> aa;
    private String serial = "";
    private Handler handler = null;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.gettingstarted_yocto_gps);
        Spinner my_spin = (Spinner) findViewById(R.id.spinner1);
        my_spin.setOnItemSelectedListener(this);
        aa = new ArrayAdapter<String>(this, android.R.layout.simple_spinner_item);
        aa.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
        my_spin.setAdapter(aa);
        handler = new Handler();
    }

    @Override
    protected void onStart() {
        super.onStart();
        try {
            aa.clear();
            YAPI.EnableUSBHost(this);
            YAPI.RegisterHub("usb");
            YModule module = YModule.FirstModule();
            while (module != null) {
                if (module.get_productName().equals("Yocto-GPS")) {
                    String serial = module.get_serialNumber();
                    aa.add(serial);
                }
                module = module.nextModule();
            }
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
        aa.notifyDataSetChanged();
        handler.postDelayed(r, 500);
    }

    @Override
    protected void onStop() {
        super.onStop();
        handler.removeCallbacks(r);
        YAPI.FreeAPI();
    }

    @Override
```

```
    public void onItemSelected(AdapterView<?> parent, View view, int pos, long id) {
        serial = parent.getItemAtPosition(pos).toString();
    }

    @Override
    public void onNothingSelected(AdapterView<?> arg0) {
    }

    final Runnable r = new Runnable() {
        public void run() {
            if (serial != null) {

                YGps gps = YGps.FindGps(serial);
                try {
                    TextView state = (TextView) findViewById(R.id.state);
                    TextView latitude = (TextView) findViewById(R.id.latitude);
                    TextView longitude = (TextView) findViewById(R.id.longitude);
                    if (gps.get_isFixed() == YGps.ISFIXED_TRUE) {
                        state.setText(String.format("%d satellites", gps.get_satCount()));
                        latitude.setText(gps.get_latitude());
                        longitude.setText(gps.get_longitude());
                    } else {
                        state.setText("fixing...");
                        latitude.setText("");
                        longitude.setText("");
                    }
                } catch (YAPI_Exception e) {
                    e.printStackTrace();
                }
            }
            handler.postDelayed(this, 1000);
        }
    };
}
```

## 17.6. Control of the module part

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

```
package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemSelectedListener;
import android.widget.ArrayAdapter;
import android.widget.Spinner;
import android.widget.Switch;
import android.widget.TextView;

import com.yoctopuce.YoctoAPI.YAPI;
import com.yoctopuce.YoctoAPI.YAPI_Exception;
import com.yoctopuce.YoctoAPI.YModule;

public class ModuleControl extends Activity implements OnItemSelectedListener
{

    private ArrayAdapter<String> aa;
    private YModule module = null;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.modulecontrol);
        Spinner my_spin = (Spinner) findViewById(R.id.spinner1);
        my_spin.setOnItemSelectedListener(this);
        aa = new ArrayAdapter<String>(this, android.R.layout.simple_spinner_item);
        aa.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
        my_spin.setAdapter(aa);
    }
```

```
    @Override
    protected void onStart()
    {
        super.onStart();

        try {
            aa.clear();
            YAPI.EnableUSBHost(this);
            YAPI.RegisterHub("usb");
            YModule r = YModule.FirstModule();
            while (r != null) {
                String hwid = r.get_hardwareId();
                aa.add(hwid);
                r = r.nextModule();
            }
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
        // refresh Spinner with detected relay
        aa.notifyDataSetChanged();
    }

    @Override
    protected void onStop()
    {
        super.onStop();
        YAPI.FreeAPI();
    }

    private void DisplayModuleInfo()
    {
        TextView field;
        if (module == null)
            return;
        try {
            field = (TextView) findViewById(R.id.serialfield);
            field.setText(module.getSerialNumber());
            field = (TextView) findViewById(R.id.logicalnamefield);
            field.setText(module.getLogicalName());
            field = (TextView) findViewById(R.id.luminosityfield);
            field.setText(String.format("%d%%", module.getLuminosity()));
            field = (TextView) findViewById(R.id.uptimefield);
            field.setText(module.getUpTime() / 1000 + " sec");
            field = (TextView) findViewById(R.id.usbcurrentfield);
            field.setText(module.getUsbCurrent() + " mA");
            Switch sw = (Switch) findViewById(R.id.beaconswitch);
            sw.setChecked(module.getBeacon() == YModule.BEACON_ON);
            field = (TextView) findViewById(R.id.logs);
            field.setText(module.get_lastLogs());

        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
    }

    @Override
    public void onItemSelected(AdapterView<?> parent, View view, int pos, long id)
    {
        String hwid = parent.getItemAtPosition(pos).toString();
        module = YModule.FindModule(hwid);
        DisplayModuleInfo();
    }

    @Override
    public void onNothingSelected(AdapterView<?> arg0)
    {
    }

    public void refreshInfo(View view)
    {
        DisplayModuleInfo();
    }

    public void toggleBeacon(View view)
    {
        if (module == null)
            return;
        boolean on = ((Switch) view).isChecked();
```

```
        try {
            if (on) {
                module.setBeacon(YModule.BEACON_ON);
            } else {
                module.setBeacon(YModule.BEACON_OFF);
            }
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
    }
}
```

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

## Changing the module settings

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

```
package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemSelectedListener;
import android.widget.ArrayAdapter;
import android.widget.EditText;
import android.widget.Spinner;
import android.widget.TextView;
import android.widget.Toast;

import com.yoctopuce.YoctoAPI.YAPI;
import com.yoctopuce.YoctoAPI.YAPI_Exception;
import com.yoctopuce.YoctoAPI.YModule;

public class SaveSettings extends Activity implements OnItemSelectedListener
{

    private ArrayAdapter<String> aa;
    private YModule module = null;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.savesettings);
        Spinner my_spin = (Spinner) findViewById(R.id.spinner1);
        my_spin.setOnItemSelectedListener(this);
        aa = new ArrayAdapter<String>(this, android.R.layout.simple_spinner_item);
        aa.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
        my_spin.setAdapter(aa);
    }

    @Override
    protected void onStart()
    {
        super.onStart();

        try {
            aa.clear();
            YAPI.EnableUSBHost(this);
            YAPI.RegisterHub("usb");
            YModule r = YModule.FirstModule();
            while (r != null) {
                String hwid = r.get_hardwareId();
```

```
                aa.add(hwid);
                r = r.nextModule();
            }
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
        // refresh Spinner with detected relay
        aa.notifyDataSetChanged();
    }

    @Override
    protected void onStop()
    {
        super.onStop();
        YAPI.FreeAPI();
    }

    private void DisplayModuleInfo()
    {
        TextView field;
        if (module == null)
            return;
        try {
            YAPI.UpdateDeviceList();// fixme
            field = (TextView) findViewById(R.id.logicalnamefield);
            field.setText(module.getLogicalName());
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
    }

    @Override
    public void onItemSelected(AdapterView<?> parent, View view, int pos, long id)
    {
        String hwid = parent.getItemAtPosition(pos).toString();
        module = YModule.FindModule(hwid);
        DisplayModuleInfo();
    }

    @Override
    public void onNothingSelected(AdapterView<?> arg0)
    {
    }

    public void saveName(View view)
    {
        if (module == null)
            return;

        EditText edit = (EditText) findViewById(R.id.newname);
        String newname = edit.getText().toString();
        try {
            if (!YAPI.CheckLogicalName(newname)) {
                Toast.makeText(getApplicationContext(), "Invalid name (" + newname + ")",
Toast.LENGTH_LONG).show();
                return;
            }
            module.set_logicalName(newname);
            module.saveToFlash(); // do not forget this
            edit.setText("");
        } catch (YAPI_Exception ex) {
            ex.printStackTrace();
        }
        DisplayModuleInfo();
    }

}
```

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

## Listing the modules

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

```
package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.util.TypedValue;
import android.view.View;
import android.widget.LinearLayout;
import android.widget.TextView;

import com.yoctopuce.YoctoAPI.YAPI;
import com.yoctopuce.YoctoAPI.YAPI_Exception;
import com.yoctopuce.YoctoAPI.YModule;

public class Inventory extends Activity
{

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.inventory);
    }

    public void refreshInventory(View view)
    {
        LinearLayout layout = (LinearLayout) findViewById(R.id.inventoryList);
        layout.removeAllViews();

        try {
            YAPI.UpdateDeviceList();
            YModule module = YModule.FirstModule();
            while (module != null) {
                String line = module.get_serialNumber() + " (" + module.get_productName() +
")";

                TextView tx = new TextView(this);
                tx.setText(line);
                tx.setTextSize(TypedValue.COMPLEX_UNIT_SP, 20);
                layout.addView(tx);
                module = module.nextModule();
            }
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
    }

    @Override
    protected void onStart()
    {
        super.onStart();
        try {
            YAPI.EnableUSBHost(this);
            YAPI.RegisterHub("usb");
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
        refreshInventory(null);
    }

    @Override
    protected void onStop()
    {
        super.onStop();
        YAPI.FreeAPI();
    }

}
```

## 17.7. Error handling

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

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

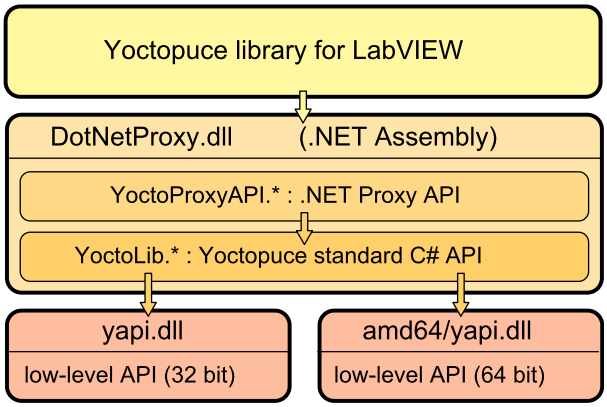
In the Java API for Android, error handling is implemented with exceptions. Therefore you must catch and handle correctly all exceptions that might be thrown by the API if you do not want your software to crash soon as you unplug a device.

# 18. Using the Yocto-GPS with LabVIEW

LabVIEW is edited by National Instruments since 1986. It is a graphic development environment: rather than writing lines of code, the users draw their programs, somewhat like a flow chart. LabVIEW was designed mostly to interface measuring tools, hence the *Virtual Instruments* name for LabVIEW programs. With visual programming, drawing complex algorithms becomes quickly fastidious. The LabVIEW Yoctopuce library was thus designed to make it as easy to use as possible. In other words, LabVIEW being an environment extremely different from other languages supported by Yoctopuce, there are major differences between the LabVIEW API and the other APIs.

## 18.1. Architecture

The LabVIEW library is based on the Yoctopuce DotNetProxy library contained in the DotNetProxyLibrary.dll DLL. In fact, it is this DotNetProxy library which takes care or most of the work by relying on the C# library which, in turn, uses the low level library coded in yapi.dll (32bits) and amd64\yapi.dll( 64bits).




*LabVIEW Yoctopuce API architecture*

You must therefore imperatively distribute the *DotNetProxyLibrary.dll*, *yapi.dll*, and *amd64\yapi.dll* with your LabVIEW applications using the Yoctopuce API.

If need be, you can find the low level API sources in the C# library and the *DotNetProxyLibrary.dll* sources in the *DotNetProxy* library.

## 18.2. Compatibility

### Firmware

For the LabVIEW Yoctopuce library to work correctly with your Yoctopuce modules, these modules need to have firmware 37120, or higher.

### LabVIEW for Linux and MacOS

At the time of writing, the LabVIEW Yoctopuce API has been tested under Windows only. It is therefore most likely that it simply does not work with the Linux and MacOS versions of LabVIEW.

### LabVIEW NXG

The LabVIEW Yoctopuce library uses many techniques which are not yet available in the new generation of LabVIEW. The library is therefore absolutely not compatible with LabVIEW NXG.

### About DotNewProxyLibrary.dll

In order to be compatible with as many versions of Windows as possible, including Windows XP, the *DotNetProxyLibrary.dll* library is compiled in .NET 3.5, which is available by default on all the Windows versions since XP.

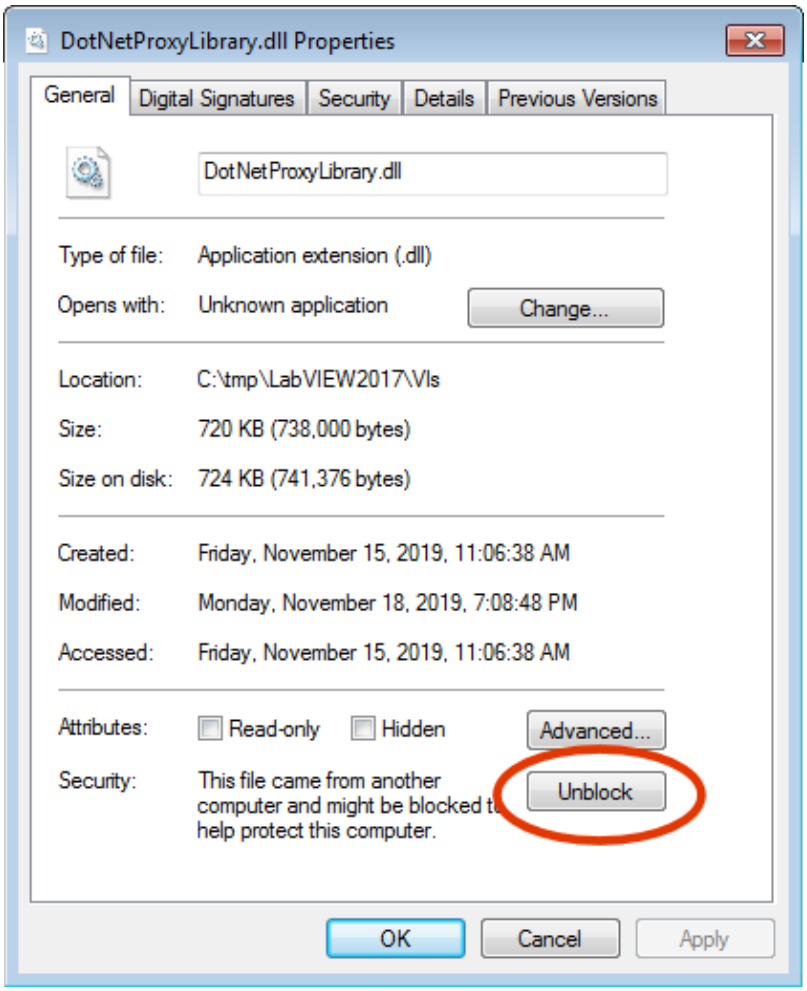
## 18.3. Installation

Download the LabVIEW library from the Yoctopuce web site[1]. It is a ZIP file in which there is a distinct directory for each version of LabVIEW. Each of these directories contains two subdirectories: the first one contains programming examples for each Yoctopuce product; the second one, called *VIs*, contains all the VIs of the API and the required DLLs.

Depending on Windows configuration and the method used to copy the *DotNetProxyLibrary.dll* on your system, Windows may block it because it comes from an other computer. This may happen when the library zip file is uncompressed with Window's file explorer. If the DLL is blocked, LabVIEW will not be able to load it and an error 1386 will occur whenever any of the Yoctopuce VIs is executed.

There are two ways to fix this. The simplest is to unblock the file with the Windows file explorer: *right click / properties* on the *DotNetProxyLibrary.dll* file, and click on the *unblock* button. But this has to be done each time a new version of the DLL is copied on your system.

[1] http://www.yoctopuce.com/EN/libraries.php




*Unblock the DotNetProxyLibrary DLL.*

Alternatively, one can modify the LabVIEW configuration by creating, in the same directory as the labview.exe executable, an XML file called *labview.exe.config* containing the following code:

```
<?xml version ="1.0"?>
<configuration>
 <runtime>
 <loadFromRemoteSources enabled="true" />
 </runtime>
</configuration>
```

Make sure to select the correct directory depending on the LabVIEW version you are using (32 bits vs. 64 bits). You can find more information about this file on the National Instruments web site.[2]

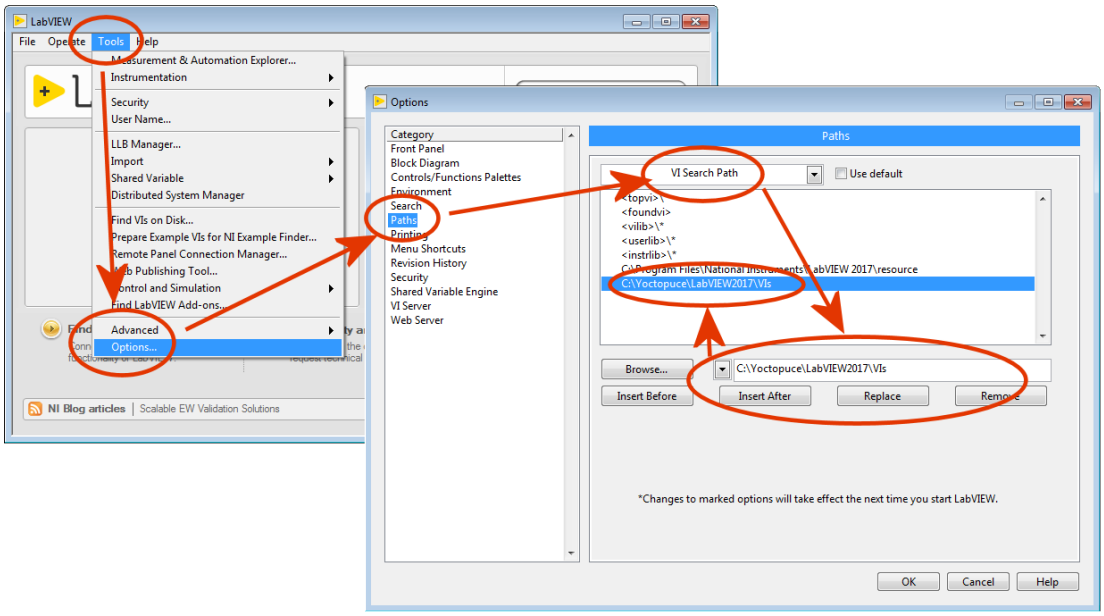
To install the LabVIEW Yoctopuce API, there are several methods.

## Method 1 : "Take-out" installation

The simplest way to use the Yoctopuce library is to copy the content of the *VIs* directory wherever you want and to use the VIs in LabVIEW with a simple drag-n-drop operation.

To use the examples provided with the API, it is simpler if you add the directory of Yoctopuce VIs into the list of where LabVIEW must look for VIs that it has not found. You can access this list through the *Tools* > *Options* > *Paths* > *VI Search Path* menu.

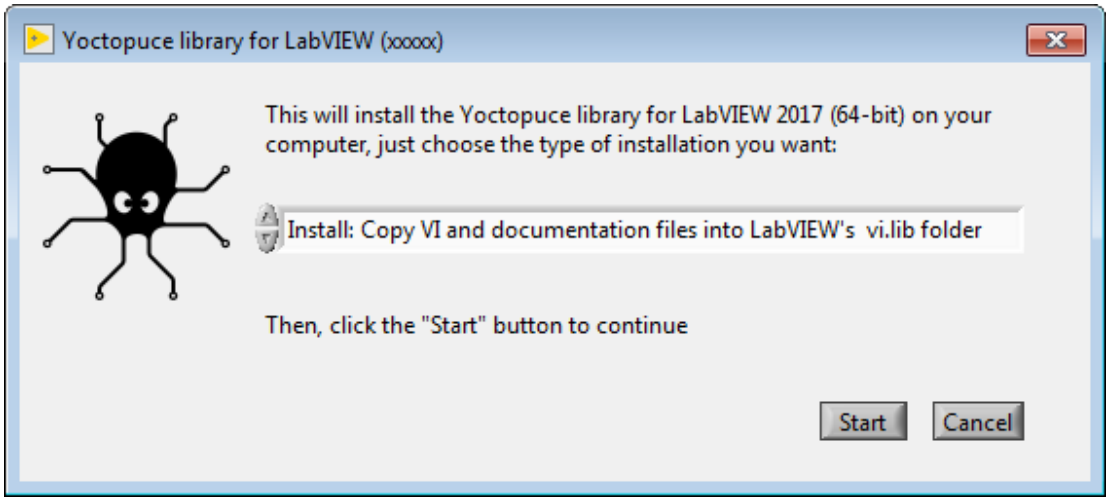
[2] https://knowledge.ni.com/KnowledgeArticleDetails?id=kA00Z000000P8XnSAK




*Configuring the "VI Search Path"*

## Method 2 : Provided installer

In each LabVIEW folder of the Library, you will find a VI named "*Install.vi*", just open the one matching your LabVIEW version.




*The provider installer*

This installer provide 3 installation options:

**Install: Keep VI and documentation files where they are.**
With this option, VI files are keep in the place where the library has been unzipped. So you will have to make sure these files are not deleted as long as you need them. Here is what the installer will do if that option is chosen:

- All references to Yoctopuce any library paths will be removed from the *viSearchPath* option in the *labview.ini* file.
- A dir.mnu palette file referring to VIs in the install folder will be created in *C:\Program Files xx\National Instruments\LabVIEW 20xx\vi.lib\addons\Yoctopuce*
- A reference to the VIs source install path will inserted into the *viSearchPath* option in the *labview.ini* file.

**Install: Copy VI and documentation files into LabVIEW's vi.lib folder**
In that case all required files are copied inside the LabVIEW's installation folder, so you will be able to delete the installation folder once the original installation is complete. Note that programming examples won't be copied. Here is the exact behaviour of the installer in that case:

- All references to Yoctopuce library paths will be removed from *viSearchPath* in *labview.ini* file
- All VIs, DLLs, and documentation files will be copied into:
  *C:\Program Files xx\National Instruments\LabVIEW 20xx\vi.lib\Yoctopuce*
- VIs will be patched with the path to copied documentation files
- A dir.mnu palette file referring to copied VIs will be created in
  *C:\Program Files xx\National Instruments\LabVIEW 20xx\vi.lib\addons\Yoctopuce*

**Uninstall Yoctopuce Library**
this option is meant to remove the LabVIEW library from your LabVIEW installation, here is how it is done:

- All references to Yoctopuce library paths will be removed from *viSearchPath* in *labview.ini* file
- Following folders, if exists, will be removed:
  *C:\Program Files xx\National Instruments\LabVIEW 20xx\vi.lib\addons\Yoctopuce*
  *C:\Program Files xx\National Instruments\LabVIEW 20xx\vi.lib\Yoctopuce*

In any case, if the *labview.ini* file needs to be modified, a backup copy will be made beforehand.

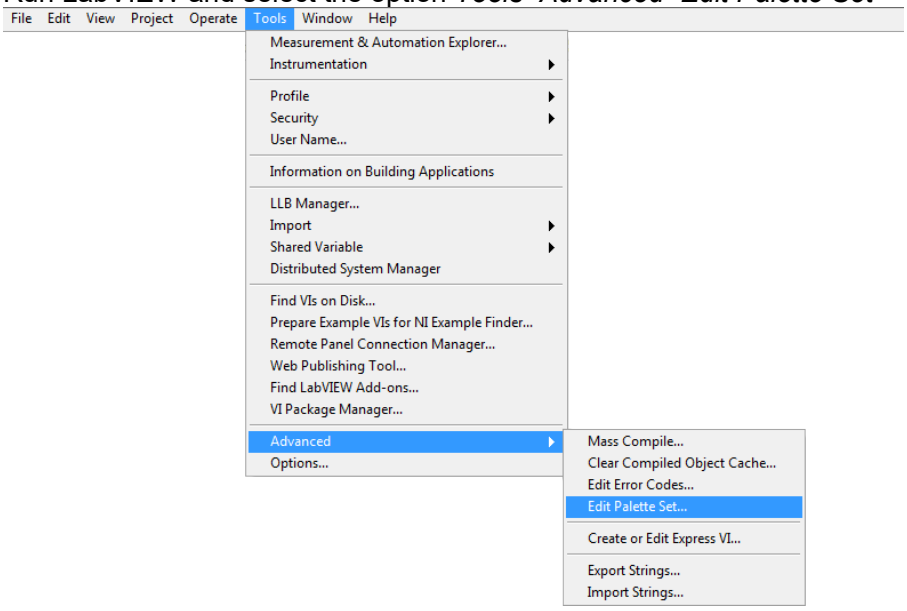
The installer identifies Yoctopuce VIs library folders by checking the presence of the YRegisterHub.vi file in said folders.

Once the installation is complete, a Yoctopuce palette will appear in *Functions/Addons* menu.

## Method 3 : Installation in a LabVIEW palette (ancillary method)

The steps to manually install the VIs directly in the LabVIEW palette are somewhat more complex. You can find the detailed procedure on the National Instruments web site [3], but here is a summary:

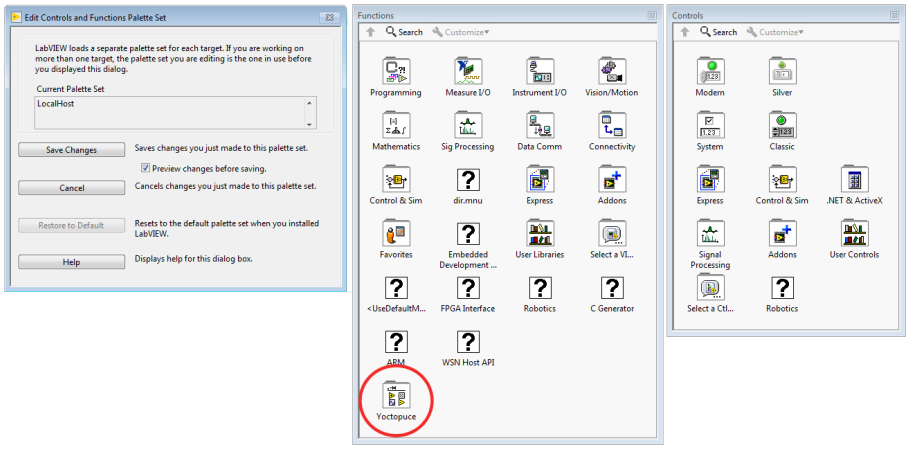
1. Create a *Yoctopuce/API* directory in the *C:\Program Files\National Instruments\LabVIEW xxxx \vi.lib* directory and copy all the VIs and DLLs of the *VIs* directory into it.
2. Create a *Yoctopuce* directory in the *C:\Program Files\National Instruments\LabVIEW xxxx \menus\Categories* directory.
3. Run LabVIEW and select the option *Tools>Advanced>Edit Palette Set*




[3] https://forums.ni.com/t5/Developer-Center-Resources/Creating-a-LabVIEW-Palette/ta-p/3520557

Three windows pop up:

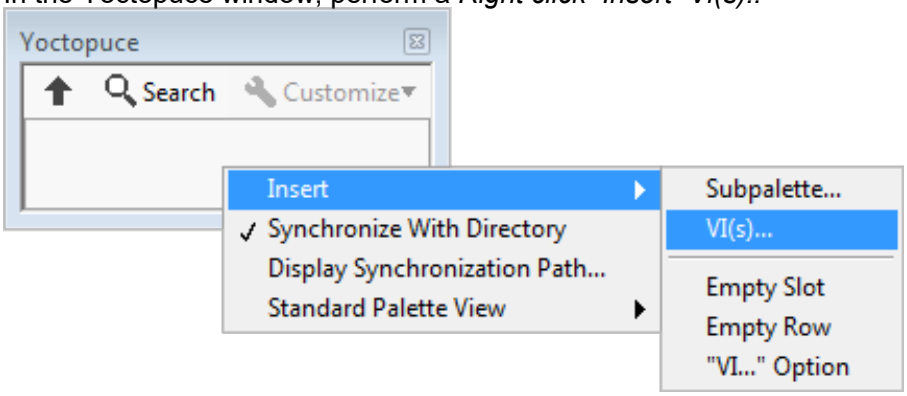
- "Edit Controls and Functions Palette Set"
- "Functions"
- "Controls"



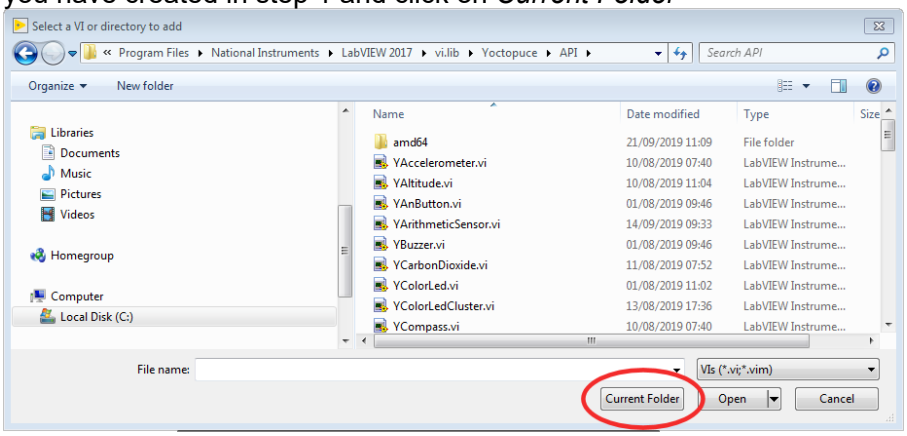

.

In the *Function* window, there is a *Yoctopuce* icon. Double-click it to create an empty "Yoctopuce" window.

4. In the Yoctopuce window, perform a *Right click>Insert>Vi(s)..*

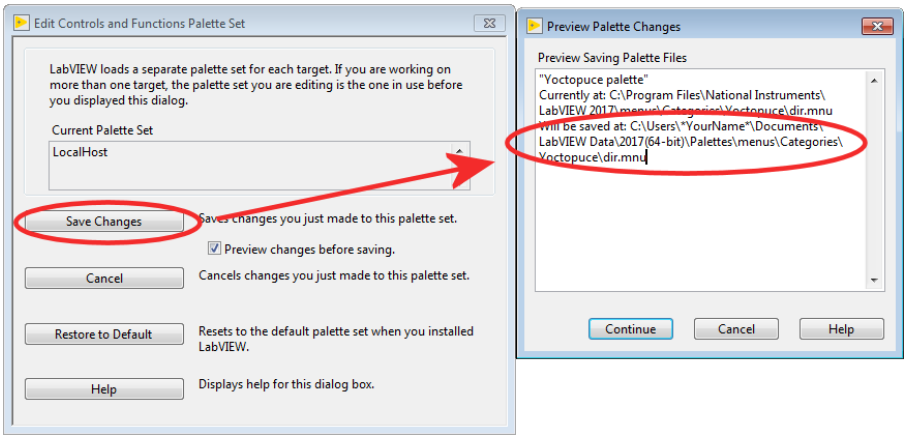



in order to open a file chooser. Put the file chooser in the *vi.lib\Yoctopuce\API* directory that you have created in step 1 and click on *Current Folder*



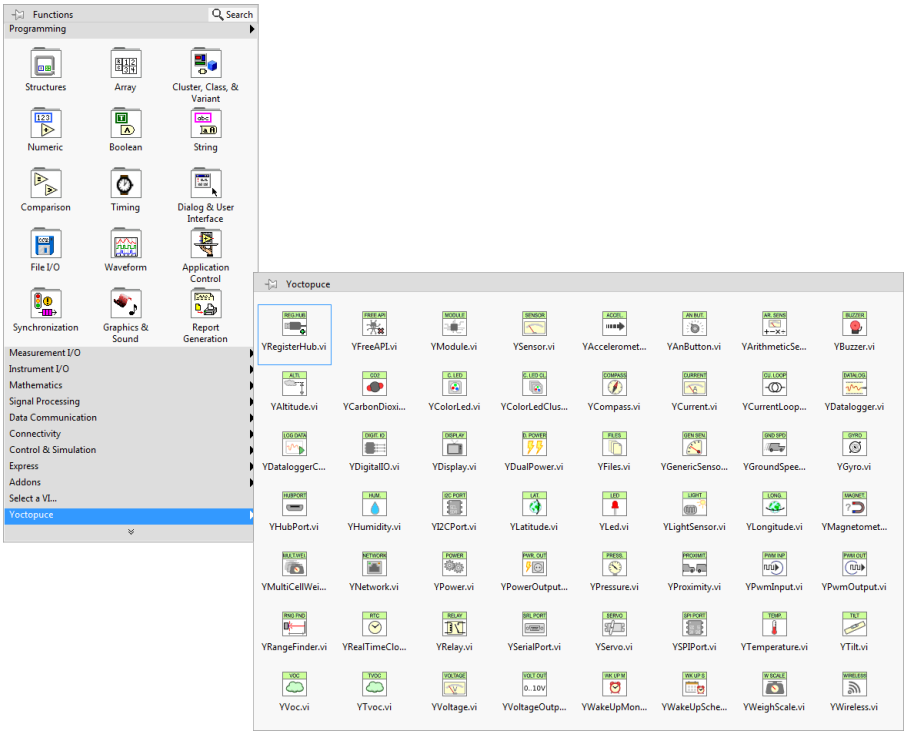

All the Yoctopuce VIs now appear in the Yoctopuce window. By default, they are sorted by alphabetical order, but you can arrange them as you see fit by moving them around with the mouse. For the palette to be easy to use, we recommend to reorganize the icons over 8 columns.

5. In the "*Edit Controls and Functions Palette Set*" window, click on the "Save Changes" button, the window indicates that it has created a `dir.mnu` file in your `Documents` directory.




Copy this file in the "`menus\Categories\Yoctopuce`" directory that you have created in step 2.

6. Restart LabVIEW, the LabVIEW palette now contains a Yoctopuce sub-palette with all the VIs of the API.

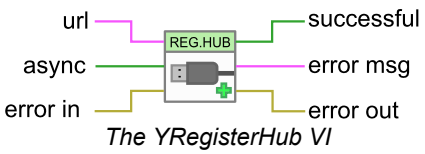



# 18.4. Presentation of Yoctopuce VIs

The LabVIEW Yoctopuce library contains one VI per class of the Yoctopuce API, as well as a few special VIs. All the VIs have the traditional connectors *Error IN* and *Error Out*.

## YRegisterHub

The `YRegisterHub` VI is used to initialize the API. You must imperatively call this VI once before you do anything in relation with Yoctopuce modules.




*The YRegisterHub VI*

The `YRegisterHub` VI takes a *url* parameter which can be:

- The `"usb"` character string to indicated that you wish to work with local modules, directly connected by USB
- An IP address to indicate that you wish to work with modules which are available through a network connection. This IP address can be that of a YoctoHub[4] or even that of a machine on which the VirtualHub[5] application is running.

In the case of an IP address, the `YRegisterHub` VI tries to contact this address and generates and error if it does not succeed, unless the *async* parameter is set to TRUE. If *async* is set to TRUE, no error is generated and Yoctopuce modules corresponding to that IP address become automatically available as soon as the said machine can be reached.

If everything went well, the *successful* output contains the value TRUE. In the opposite case, it contains the value FALSE and the *error msg* output contains a string of characters with a description of the error.

You can use several `YRegisterHub` VIs with distinct URLs if you so wish. However, on the same machine, there can be only one process accessing local Yoctopuce modules directly by USB (*url* set to `"usb"`). You can easily work around this limitation by running the VirtualHub software on the local machine and using the `"127.0.0.1"` url.

## YFreeAPI

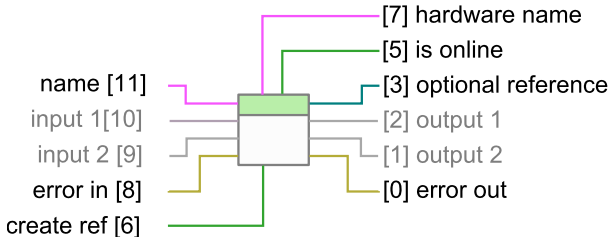
The `YFreeAPI` VI enables you to free resources allocated by the Yoctopuce API.


*The YFreeAPI VI*

You must call the `YFreeAPI` VI when your code is done with the Yoctopuce API. Otherwise, direct USB access (*url* set to `"usb"`) could stay locked after the execution of your VI, and stay so for as long as LabVIEW is not completely closed.

## Structure of the VIs corresponding to a class

The other VIs correspond to each function/class of the Yoctopuce API, they all have the same structure:




*Structure of most VIs of the API.*

- Connector [11]: *name* must contain the hardware name or the logical name of the intended function.
- Connectors [10] and [9]: input parameters depending on the nature of the VI.
- Connectors [8] and [0] : *error in* and *error out*.
- Connector [7] : Unique hardware name of the found function.
- Connector [5] : *is online* contains TRUE if the function is available, FALSE otherwise.
- Connectors [2] and [1]: output values depending on the nature of the VI.
- Connector [6]: If this input is set to TRUE, connector [3] contains a reference to the *Proxy* objects implemented by the VI[6]. This input is initialized to FALSE by default.

[4] www.yoctopuce.com/EN/products/category/extensions-and-networking
[5] http://www.yoctopuce.com/EN/virtualhub.php
[6] see section *Using Proxy objects*

- Connector [3]: Reference on the *Proxy* object implemented by the VI if input [6] is TRUE. This object enables you to access additional features.

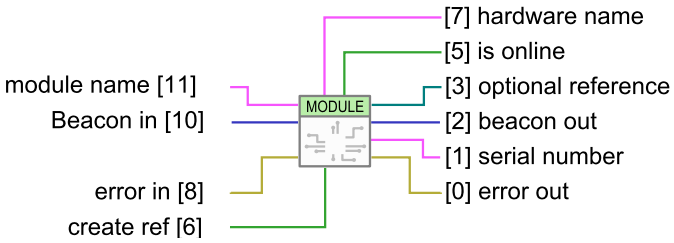
You can find the list of functions available on your Yocto-GPS in chapter *Programming, general concepts*.

If the desired function (parameter *name*) is not available, this does not generate an error, but the *is online* output contains FALSE and all the other outputs contain the value "N/A" whenever possible. If the desired function becomes available later in the life of your program, *is online* switches to TRUE automatically.

If the *name* parameter contains an empty string, the VI targets the first available function of the same type. If no function is available, *is online* is set to FALSE.

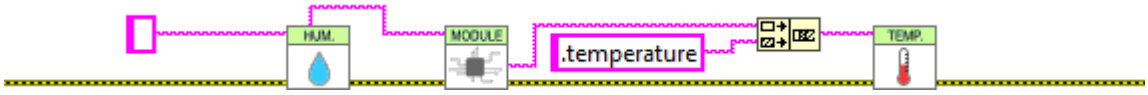
## The YModule VI

The `YModule` VI enables you to interface with the "module" section of each Yoctopuce module. It enables you to drive the module led and to know the serial number of the module.




*The YModule VI*

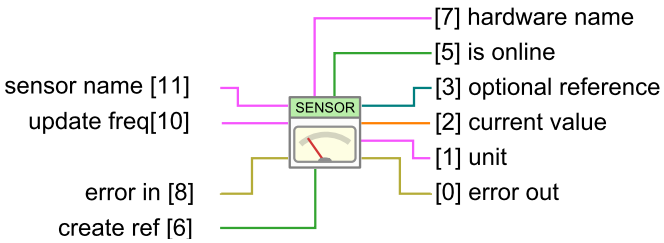
The *name* input works slightly differently from other VIs. If it is called with a *name* parameter corresponding to a function name, the `YModule` VI finds the *Module* function of the module hosting the function. You can therefore easily find the serial number of the module of any function. This enables you to build the name of other functions which are located on the same module. The following example finds the first available *YHumidity* function and builds the name of the *YTemperature* function located on the same module. The examples provided with the Yoctopuce API make extensive use of this technique.




*Using the YModule VI to retrieve functions hosted on the same module*

## The sensor VIs

All the VIs corresponding to Yoctopuce sensors have exactly the same geometry. Both outputs enable you to retrieve the value measured by the corresponding sensor as well the unit used.



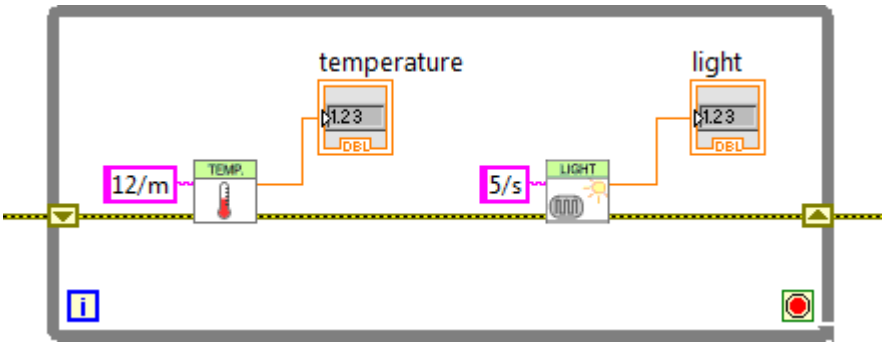

*The sensor VIs have all exactly the same geometry*

The *update freq* input parameter is a character string enabling you to configure the way in which the output value is updated:

- "auto" : The VI value is updated as soon as the sensor detects a significant modification of the value. It is the default behavior.
- "x/s": The VI value is updated *x* times per second with the current value of the sensor.

- "x/m","x/h": The VI value is updated *x* times per minute (resp. hour) with the average value over the latest period. Note, maximum frequencies are (60/m) and (3600/h), for higher frequencies use the (x/s) syntax.

The update frequency of the VI is a parameter managed by the physical Yoctopuce module. If several VIs try to change the frequency of the same sensor, the valid configuration is that of the latest call. It is however possible to set different update frequencies to different sensors on the same Yoctopuce module.

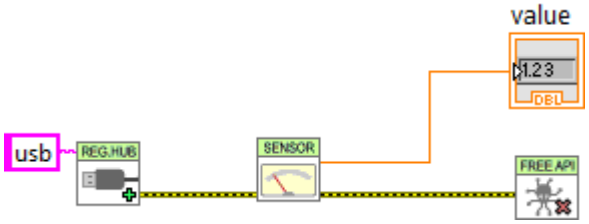



*Changing the update frequency of the same module*

The update frequency of the VI is completely independent from the sampling frequency of the sensor, which you usually cannot modify. It is useless and counterproductive to define an update frequency higher than the sensor sampling frequency.

## 18.5. Functioning and use of VIs

Here is one of the simplest example of VIs using the Yoctopuce API.

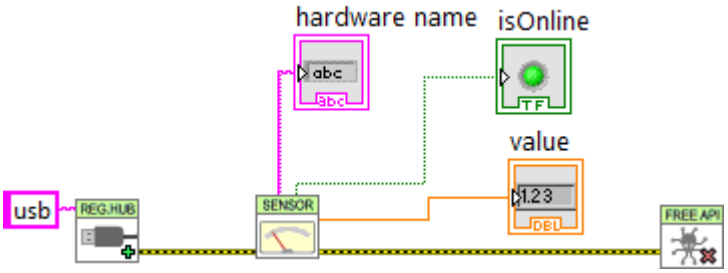



*Minimal example of use of the LabVIEW Yoctopuce API*

This example is based on the `YSensor` VI which is a generic VI enabling you to interface any sensor function of a Yoctopuce module. You can replace this VI by any other from the Yoctopuce API, they all have the same geometry and work in the same way. This example is limited to three actions:

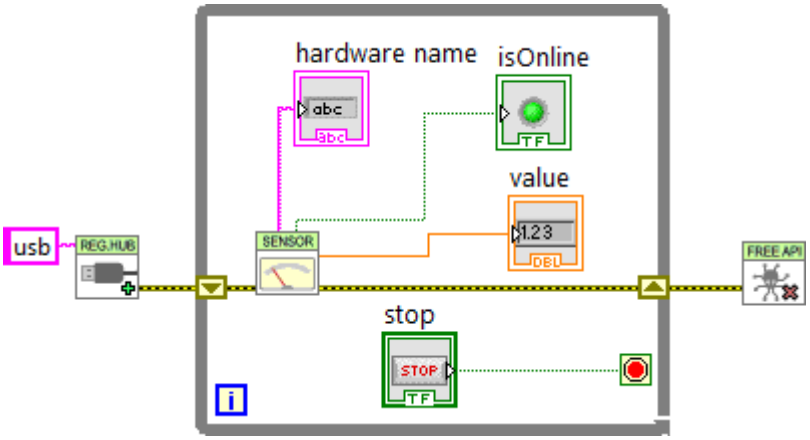
1. It initializes the API in native ("usb") mode with the `YRegisterHub` VI.
2. It displays the value of the first Yoctopuce sensor it finds thanks to the `YSensor` VI.
3. It frees the API thanks to the `YFreeAPI` VI.

This example automatically looks for an available sensor. If there is such a sensor, we can retrieve its name through the *hardware name* output and the *isOnline* output equals TRUE. If there is no available sensor, the VI does not generate an error but emulates a ghost sensor which is "offline". However, if later in the life of the application, a sensor becomes available because it has been connected, *isOnline* switches to TRUE and the *hardware name* contains the name of the sensor. We can therefore easily add a few indicators in the previous example to know how the executions goes.




*Use of the hardware name and isOnline outputs*

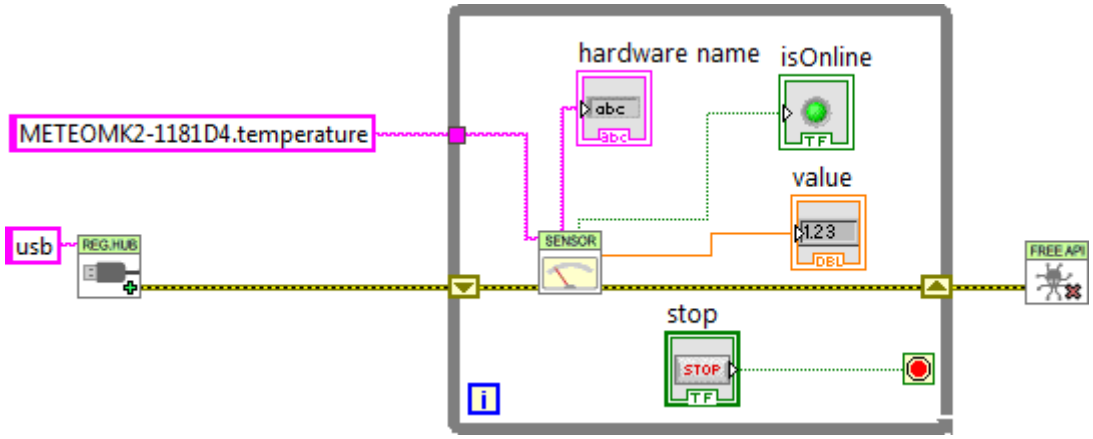
The VIs of the Yoctopuce API are actually an entry door into the library. Internally, this mechanism works independently of the Yoctopuce VIs. Indeed, most communications with electronic modules are managed automatically as background tasks. Therefore, you do not necessarily need to take any specific care to use Yoctopuce VIs, you can for example use them in a non-delayed loop without creating any specific problem for the API.




*The Yoctopuce VIs can be used in a non-delayed loop*

Note that the `YRegisterHub` VI is not inside the loop. The `YRegisterHub` VI is used to initialize the API. Unless you have several URLs that you need to register, it is better to call the `YRegisterHub` VI only once.

When the *name* parameter is initialized to an empty string, the Yoctopuce VIs automatically look for a function they can work with. This is very handy when you know that there is only one function of the same type available and when you do not want to manage its name. If the *name* parameter contains a hardware name or a logical name, the VI looks for the corresponding function. If it does not find it, it emulates an *offline* function while it waits for the true function to become available.

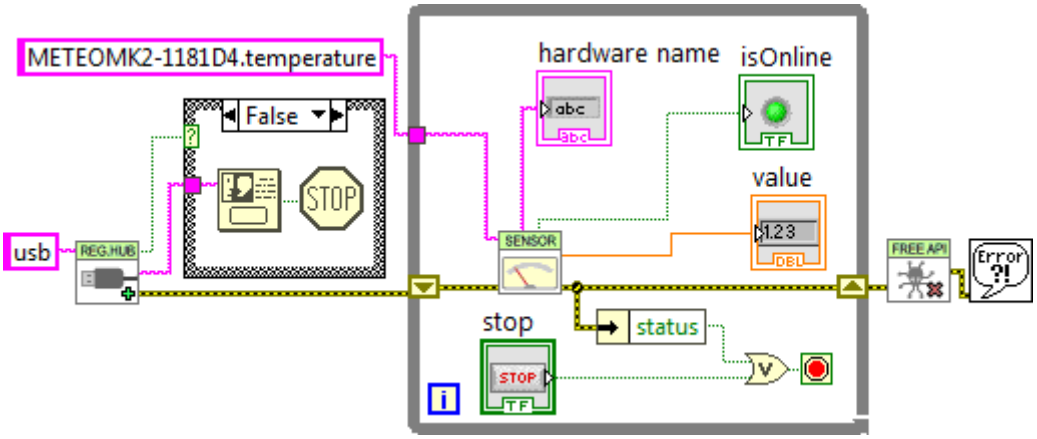



*Using names to identify the functions to be used*

### Error handling

The LabVIEW Yoctopuce API is coded to handle errors as smoothly as possible: for example, if you use a VI to access a function which does not exist, the *isOnline* output is set to FALSE, the other outputs are set to *NaN*, and thus the inputs do not have any impact. Fatal errors are propagated through the traditional *error in*, *error out* channel.

However, the `YRegisterHub` VI manages connection errors slightly differently. In order to make them easier to manage, connection errors are signaled with *Success* and *error msg* outputs. If there is an issue during a call to the `YRegisterHub` VI, *Success* contains FALSE and *error msg* contains a description of the error.



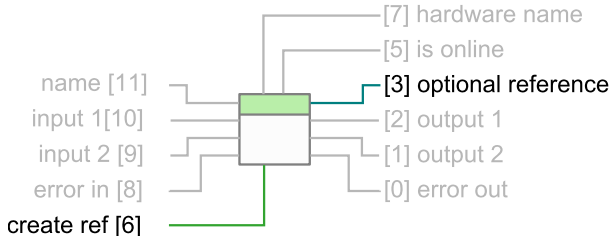

*Error handling*

The most common error message is "*Another process is already using yAPI*". It means that another application, LabVIEW or other, already uses the API in native USB mode. For technical reasons, the native USB API can be used by only one application at the same time on the same machine. You can easily work around this limitation by using the network mode.

## 18.6. Using *Proxy* objects

The Yoctopuce API contains hundreds of methods, functions, and properties. It was not possible, or desirable, to create a VI for each of them. Therefore, there is a VI per class that shows the two properties that Yoctopuce deemed the most useful, but this does not mean that the rest is not available.

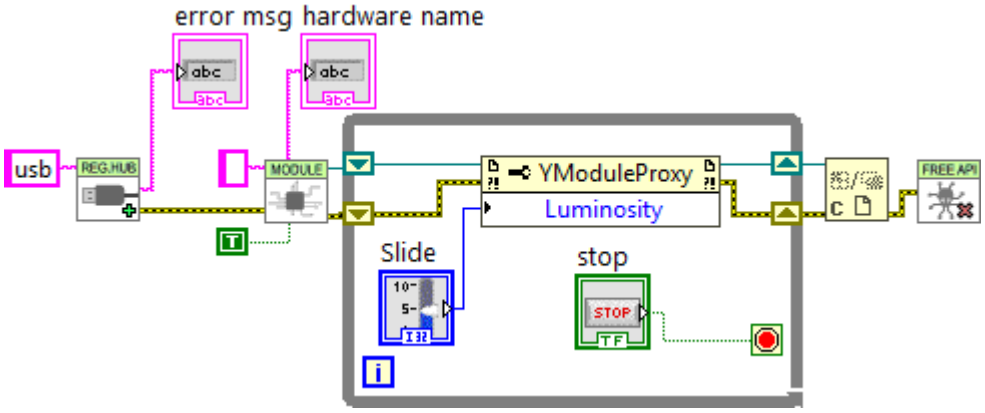
Each VI corresponding to a class has two connectors *create ref* and *optional ref* which enable you to obtain a reference on the *Proxy* object of the *.NET Proxy* API on which the LabVIEW library is built.




*The connectors to obtain a reference on the Proxy object corresponding to the VI*

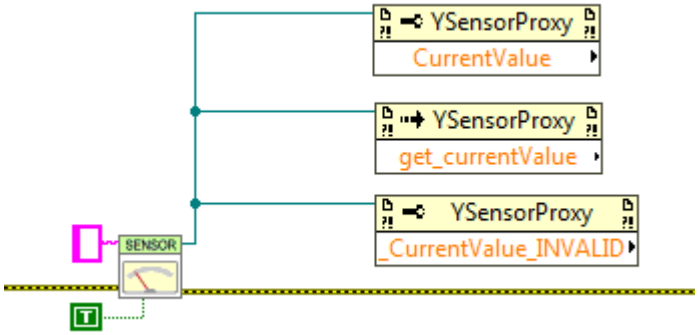
To obtain this reference, you only need to set *optional ref* to TRUE. Note, it is essential to close all references created in this way, otherwise you risk to quickly saturate the computer memory.

Here is an example which uses this technique to change the luminosity of the leds of a Yoctopuce module.




*Regulating the luminosity of the leds of a module*

Note that each reference allows you to obtain properties (*property* nodes) as well as methods (*invoke* nodes). By convention, properties are optimized to generate a minimum of communication with the modules. Therefore, we recommend to use them rather than the corresponding *get_xxx* and *set_xxx* methods which might seem equivalent but which are not optimized. Properties also enable you to retrieve the various constants of the API, prefixed with the "_" character. For technical reasons, the *get_xxx* and *set_xxx* methods are not all available as properties.

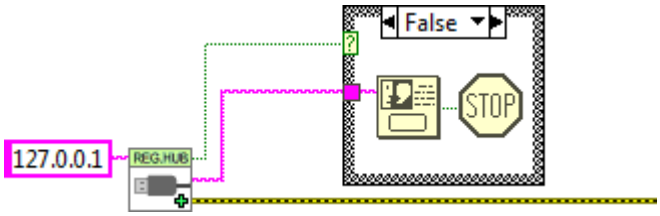



*Property and Invoke nodes: Using properties, methods and constants*

You can find a description of all the available properties, functions, and methods in the documentation of the *.NET Proxy* API.

## Network mode

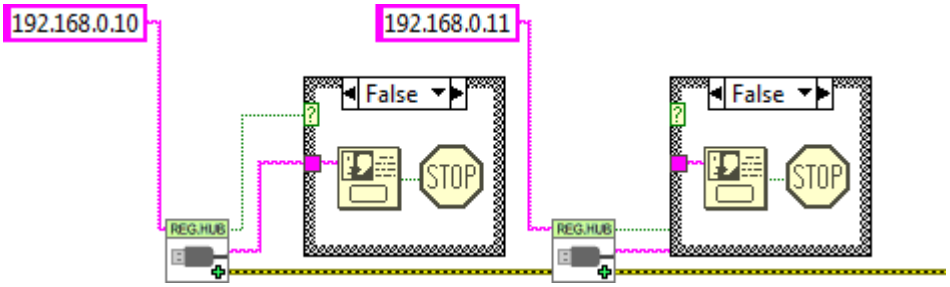
On a given machine, there can be only one process accessing local Yoctopuce modules directly by USB (url set to `"usb"`). It is however possible that multiple process connect in parallel to YoctoHubs[7] or tp a machine on which *VirtualHub*[8] is running, including the local machine. Therefore, if you use the local address of your machine (127.0.0.1) and if a VirtualHub runs on it, you can work around the limitation which prevents using the native USB API in parallel.




*Network mode*

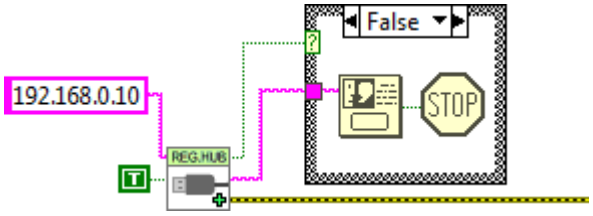
[7] https://www.yoctopuce.com/EN/products/category/extensions-and-networking
[8] www.yoctopuce.com/EN/virtualhub.php

In the same way, there is no limitation on the number of network interfaces to which the API can connect itself in parallel. This means that it is quite possible to make multiple calls to the `YRegisterHub` VI. This is the only case where it is useful to call the `YRegisterHub` VI several times in the life of the application.




*You can have multiple network connections*

By default, the `YRegisterHub` VI tries to connect itself on the address given as parameter and generates an error (*success*=FALSE) when it cannot do so because nobody answers. But if the *async* parameter is initialized to TRUE, no error is generated when the connection does not succeed. If the connection becomes possible later in the life of the application, the corresponding modules are automatically made available.



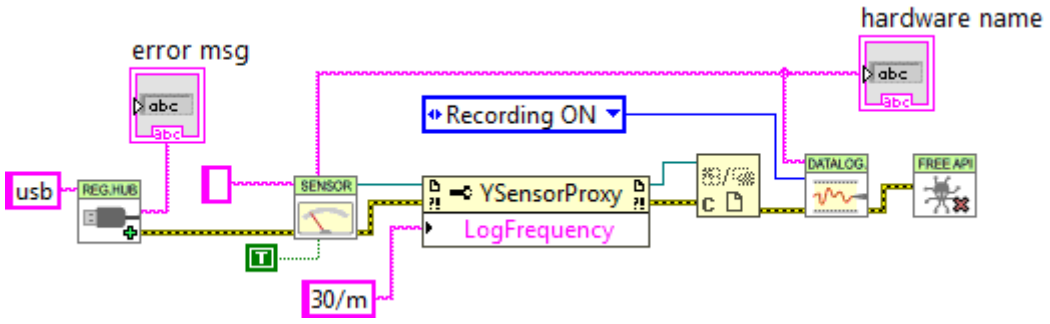

*Asynchronous connection*

# 18.7. Managing the data logger

Almost all the Yoctopuce sensors have a data logger which enables you to store the measures of the sensors in the non-volatile memory of the module. You can configure the data logger with the VirtualHub, but also with a little bit of LabVIEW code.

## Logging

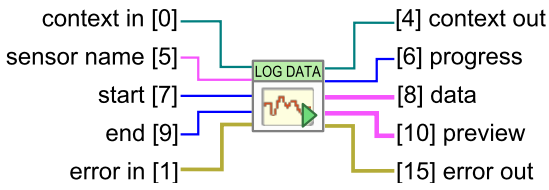
To do so, you must configure the logging frequency by using the "LogFrequency" property which you can reach with a reference on the *Proxy* object of the sensor you are using. Then, you must turn the data logger on thanks to the `YDataLogger` VI. Note that, like with the `YModule` VI, you can obtain the `YDataLogger` VI corresponding to a module with its own name, but also with the name of any of the functions available on the same module.




*Activating the data logger*

## Reading

You can retrieve the data in the data logger with the `YDataLoggerContents` VI.

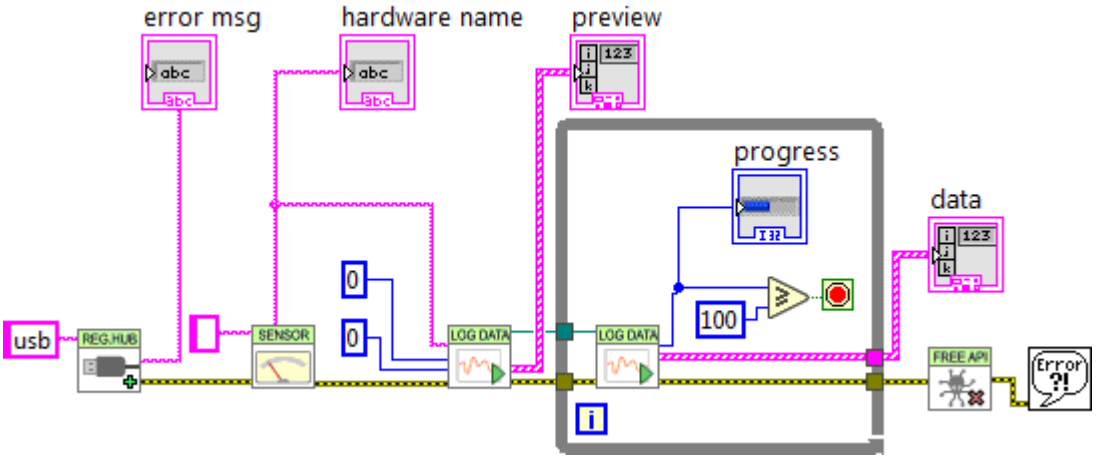



*The YDataLoggerContents VI*

Retrieving the data from the logger of a Yoctopuce module is a slow process which can take up to several tens of seconds. Therefore, we designed the VI enabling this operation to work iteratively.

As a first step, you must call the VI with a sensor name, a start date, and an end date (UTC UNIX timestamp). The (0,0) pair enables you to obtain the complete content of the data logger. This first call enables you to obtain a summary of the data logger content and a context.

As a second step, you must call the *YDataLoggerContents* VI in a loop with the context parameter, until the *progress* output reaches the 100 value. At this time, the data output represents the content of the data logger.




*Retrieving the content of the data logger*

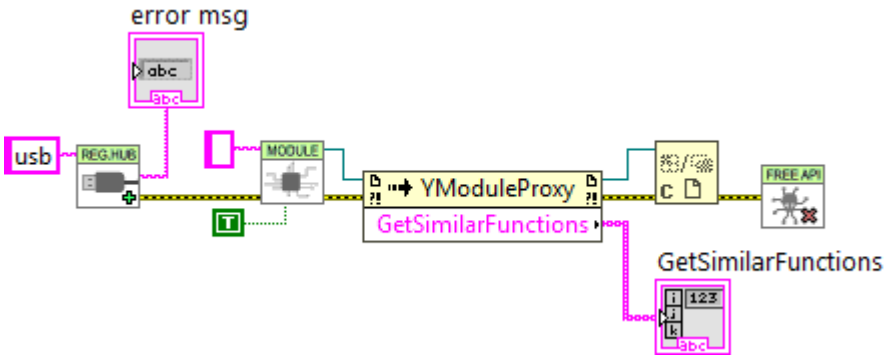
The results and the summary are returned as an array of structures containing the following fields:

- *startTime*: beginning of the measuring period
- *endTime*: end of the measuring period
- *averageValue*: average value for the period
- *minValue*: minimum value over the period
- *maxValue*: maximum value over the period

Note that if the logging frequency is superior to 1Hz, the data logger stores only current values. In this case, *averageValue*, *minValue*, and *maxValue* share the same value.

# 18.8. Function list

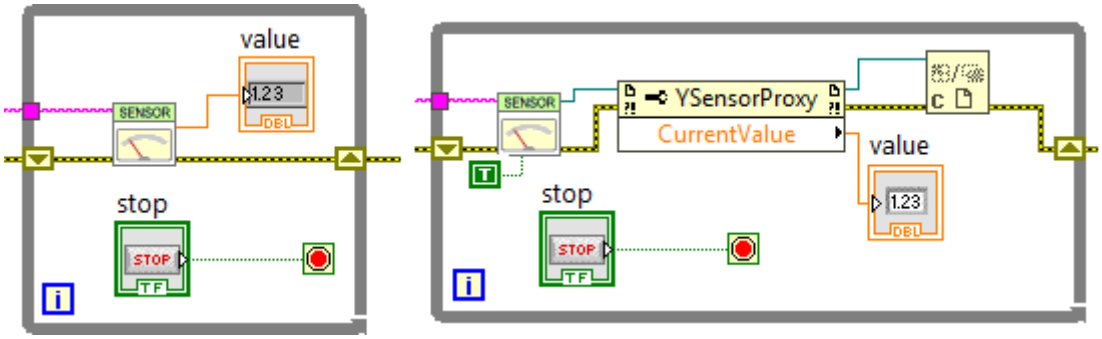
Each VI corresponding to an object of the *Proxy API* enables you to list all the functions of the same class with the *getSimilarfunctions()* method of the corresponding *Proxy* object. Thus, you can easily perform an inventory of all the connected modules, of all the connected sensors, of all the connected relays, and so on.




*Retrieving the list of all the modules which are connected*

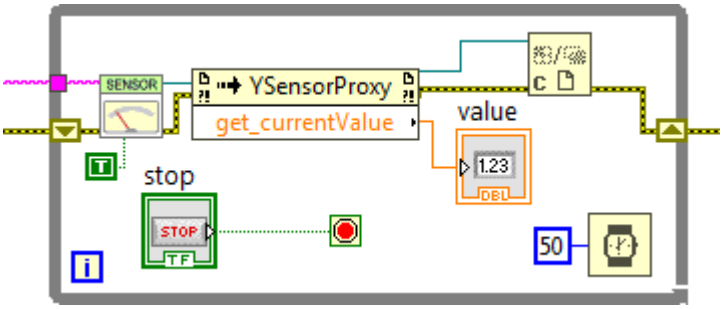
## 18.9. A word on performances

The LabVIEW Yoctopuce API is optimized so that all the VIs and *.NET Proxy* API object properties generate a minimum of communication with Yoctopuce modules. Thus, you can use them in loops without taking any specific precaution: you *do not have to* slow down the loops with a timer.




*These two loops generate little USB communication and do not need to be slowed down*

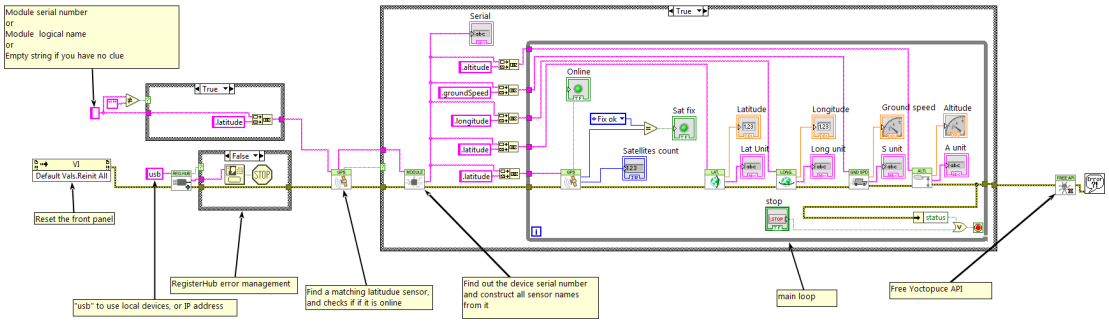
However, almost all the methods of the available Proxy objects initiate a communication with the Yoctopuce modules each time they are called. You should therefore avoid calling them too often without purpose.




*This loop, using a method, must be slowed down*

## 18.10. A full example of a LabVIEW program

Here is a short example of how to use the Yocto-GPS in LabVIEW. After a call to the *RegisterHub* VI, the *YCarbonDioxyde* VI finds the first GPS function available, then use the *YModule* VI to find out the device serial number. This number is used to build the name of all functions present on the device. Theses names are used to initialize one VI per function. This technique avoids ambiguities when several Yocto-GPS are connected at the same time. Once every VI is initialized, the sensors values can be displayed. When the application is about to exit, it frees the Yoctopuce API, thanks to the *YFreeAPI* VI.




*Example of Yocto-GPS usage in LabVIEW*

If you read this documentation on screen, you can zoom on the image above. You can also find this example in the LabVIEW Yoctopuce library.

# 18.11. Differences from other Yoctopuce APIs

Yoctopuce does everything it can to maintain a strong coherence between its different programming libraries. However, LabVIEW being clearly apart as an environment, there are, as a consequence, important differences from the other libraries.

These differences were introduced to make the use of modules as easy as possible and requiring a minimum of LabVIEW code.

## YFreeAPI

In the opposite to other languages, you must absolutely free the native API by calling the `YFreeAPI` VI when your code does not need to use the API anymore. If you forget this call, the native API risks to stay locked for the other applications until LabVIEW is completely closed.

## Properties

In the opposite to classes of the other APIs, classes available in LabVIEW implement *properties*. By convention, these properties are optimized to generate a minimum of communication with the modules while automatically refreshing. By contrast, methods of type *get_xxx* and *set_xxx* systematically generate communications with the Yoctopuce modules and must be called sparingly.

## Callback vs. Properties

There is no callback in the LabVIEW Yoctopuce API, the VIs automatically refresh: they are based on the properties of the *.NET Proxy* API objects.

# 19. Using with unsupported languages

Yoctopuce modules can be driven from most common programming languages. New languages are regularly added, depending on the interest expressed by Yoctopuce product users. Nevertheless, some languages are not, and will never be, supported by Yoctopuce. There can be several reasons for this: compilers which are not available anymore, unadapted environments, etc.

However, there are alternative methods to access Yoctopuce modules from an unsupported programming language.

## 19.1. Command line

The easiest method to drive Yoctopuce modules from an unsupported programming language is to use the command line API through system calls. The command line API is in fact made of a group of small executables which are easy to call. Their output is also easy to analyze. As most programming languages allow you to make system calls, the issue is solved with a few lines of code.

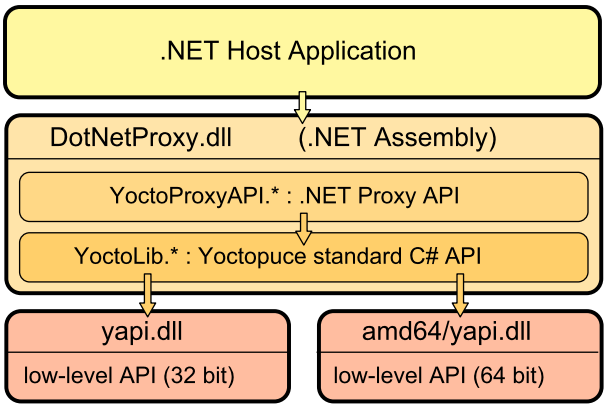
However, if the command line API is the easiest solution, it is neither the fastest nor the most efficient. For each call, the executable must initialize its own API and make an inventory of USB connected modules. This requires about one second per call.

## 19.2. .NET Assembly

A .NET Assembly enables you to share a set of pre-compiled classes to offer a service, by stating entry points which can be used by third-party applications. In our case, it's the whole Yoctopuce library which is available in the .NET Assembly, so that it can be used in any environment which supports .NET Assembly dynamic loading.

The Yoctopuce library as a .NET Assembly does not contain only the standard C# Yoctopuce library, as this wouldn't have allowed an optimal use in all environments. Indeed, we cannot expect host applications to necessarily offer a thread system or a callback system, although they are very useful to manage plug-and-play events and sensors with a high refresh rate. Likewise, we can't expect from external applications a transparent behavior in cases where a function call in Assembly creates a delay because of network communications.

Therefore, we added to it an additional layer, called *.NET Proxy* library. This additional layer offers an interface very similar to the standard library but somewhat simplified, as it internally manages all the callback mechanisms. Instead, this library offers mirror objects, called *Proxys*, which publish through *Properties* the main attributes of the Yoctopuce functions such as the current measure, configuration parameters, the state, and so on.




*.NET Assembly Architecture*

The callback mechanism automatically updates the properties of the *Proxys* objects, without the host application needing to care for it. The later can thus, at any time and without any risk of latency, display the value of all properties of Yoctopuce Proxy objects.

Pay attention to the fact that the `yapi.dll` low-level communication library is **not** included in the .NET Assembly. You must therefore keep it together with `DotNetProxyLibrary.dll`. The 32 bit version must be located in the same directory as `DotNetProxyLibrary.dll`, while the 64 bit version must be in a subdirectory `amd64`.

## Example of use with MATLAB

Here is how to load our Proxy .NET Assembly in MATLAB and how to read the value of the first sensor connected by USB found on the machine:

```
NET.addAssembly("C:/Yoctopuce/DotNetProxyLibrary.dll");
import YoctoProxyAPI.*

errmsg = YAPIProxy.RegisterHub("usb");
sensor = YSensorProxy.FindSensor("");
measure = sprintf('%.3f %s', sensor.CurrentValue, sensor.Unit);
```

## Example of use in PowerShell

PowerShell commands are a little stranger, but we can recognize the same structure:

```
Add-Type -Path "C:/Yoctopuce/DotNetProxyLibrary.dll"

$errmsg = [YoctoProxyAPI.YAPIProxy]::RegisterHub("usb")
$sensor = [YoctoProxyAPI.YSensorProxy]::FindSensor("")
$measure = "{0:n3} {1}" -f $sensor.CurrentValue, $sensor.Unit
```

## Specificities of the .NET Proxy library

With regards to classic Yoctopuce libraries, the following differences in particular should be noted:

### No FirstModule/nextModule method

To obtain an object referring to the first found module, we call `YModuleProxy.FindModule("")`. If there is no connected module, this method returns an object with its `module.IsOnline` property set to `False`. As soon as a module is connected, the property changes to `True` and the module hardware identifier is updated.

To list modules, you can call the `module.GetSimilarFunctions()` method which returns an array of character strings containing the identifiers of all the modules which were found.

### No callback function

Callback functions are implemented internally and they update the object properties. You can therefore simply poll the properties, without significant performance penalties. Be aware that if you

use one of the function that disables callbacks, the automatic refresh of object properties may not work anymore.

A new method `YAPIProxy.GetLog` makes it possible to retrieve low-level debug logs without using callbacks.

**Enumerated types**
In order to maximize compatibility with host applications, the .NET Proxy library does not use true .NET enumerated types, but simple integers. For each enumerated type, the library includes public constants named according to the possible values. Contrarily to standard Yoctopuce libraries, numeric values always start from `1`, as the value `0` is reserved to return an invalid value, for instance when the device is disconnected.

**Invalid numeric results**
For all numeric results, rather than using an arbitrary constant, the invalid value returned in case of error is *NaN*. You should therefore use function *isNaN()* to detect this value.

### Using .NET Assembly without the Proxy library

If for a reason or another you don't want to use the Proxy library, and if your environment allows it, you can use the standard C# API as it is located in the Assembly, under the `YoctoLib` namespace. Beware however not to mix both types of use: either you go through the Proxy library, or you use he `YoctoLib` version directly, but not both!

### CompatibilitÃ©

For the LabVIEW Yoctopuce library to work correctly with your Yoctopuce modules, these modules need to have firmware 37120, or higher.

In order to be compatible with as many versions of Windows as possible, including Windows XP, the *DotNetProxyLibrary.dll* library is compiled in .NET 3.5, which is available by default on all the Windows versions since XP. As of today, we have never met any non-Windows environment able to load a .NET Assembly, so we only ship the low-level communication dll for Windows together with the assembly.

## 19.3. VirtualHub and HTTP GET

The *VirtualHub* is available on almost all current platforms. It is generally used as a gateway to provide access to Yoctopuce modules from languages which prevent direct access to hardware layers of a computer (JavaScript, PHP, Java, ...).

In fact, the *VirtualHub* is a small web server able to route HTTP requests to Yoctopuce modules. This means that if you can make an HTTP request from your programming language, you can drive Yoctopuce modules, even if this language is not officially supported.

### REST interface

At a low level, the modules are driven through a REST API. Thus, to control a module, you only need to perform appropriate requests on the *VirtualHub*. By default, the *VirtualHub* HTTP port is 4444.

An important advantage of this technique is that preliminary tests are very easy to implement. You only need a *VirtualHub* and a simple web browser. If you copy the following URL in your preferred browser, while the *VirtualHub* is running, you obtain the list of the connected modules.

```
http://127.0.0.1:4444/api/services/whitePages.txt
```

Note that the result is displayed as text, but if you request *whitePages.xml*, you obtain an XML result. Likewise, *whitePages.json* allows you to obtain a JSON result. The *html* extension even allows you to display a rough interface where you can modify values in real time. The whole REST API is available in these different formats.

## Driving a module through the REST interface

Each Yoctopuce module has its own REST interface, available in several variants. Let us imagine a Yocto-GPS with the *YGNSSMK1-12345* serial number and the *myModule* logical name. The following URL allows you to know the state of the module.

```
http://127.0.0.1:4444/bySerial/YGNSSMK1-12345/api/module.txt
```

You can naturally also use the module logical name rather than its serial number.

```
http://127.0.0.1:4444/byName/myModule/api/module.txt
```

To retrieve the value of a module property, simply add the name of the property below *module*. For example, if you want to know the signposting led luminosity, send the following request:

```
http://127.0.0.1:4444/bySerial/YGNSSMK1-12345/api/module/luminosity
```

To change the value of a property, modify the corresponding attribute. Thus, to modify the luminosity, send the following request:

```
http://127.0.0.1:4444/bySerial/YGNSSMK1-12345/api/module?luminosity=100
```

## Driving the module functions through the REST interface

The module functions can be manipulated in the same way. To know the state of the latitude function, build the following URL:

```
http://127.0.0.1:4444/bySerial/YGNSSMK1-12345/api/latitude.txt
```

Note that if you can use logical names for the modules instead of their serial number, you cannot use logical names for functions. Only hardware names are authorized to access functions.

You can retrieve a module function attribute in a way rather similar to that used with the modules. For example:

```
http://127.0.0.1:4444/bySerial/YGNSSMK1-12345/api/latitude/logicalName
```

Rather logically, attributes can be modified in the same manner.

```
http://127.0.0.1:4444/bySerial/YGNSSMK1-12345/api/latitude?logicalName=myFunction
```

You can find the list of available attributes for your Yocto-GPS at the beginning of the *Programming* chapter.

## Accessing Yoctopuce data logger through the REST interface

*This section only applies to devices with a built-in data logger.*

The preview of all recorded data streams can be retrieved in JSON format using the following URL:

```
http://127.0.0.1:4444/bySerial/YGNSSMK1-12345/dataLogger.json
```

Individual measures for any given stream can be obtained by appending the desired function identifier as well as start time of the stream:

```
http://127.0.0.1:4444/bySerial/YGNSSMK1-12345/dataLogger.json?id=latitude&utc=1389801080
```

# 19.4. Using dynamic libraries

The low level Yoctopuce API is available under several formats of dynamic libraries written in C. The sources are available with the C++ API. If you use one of these low level libraries, you do not need the *VirtualHub* anymore.

| Filename | Platform |
|---|---|
| libyapi.dylib | Max OS X |
| libyapi-amd64.so | Linux Intel (64 bits) |
| libyapi-armel.so | Linux ARM EL (32 bits) |
| libyapi-armhf.so | Linux ARM HL (32 bits) |
| libyapi-aarch64.so | Linux ARM (64 bits) |
| libyapi-i386.so | Linux Intel (32 bits) |
| yapi64.dll | Windows (64 bits) |
| yapi.dll | Windows (32 bits) |

These dynamic libraries contain all the functions necessary to completely rebuild the whole high level API in any language able to integrate these libraries. This chapter nevertheless restrains itself to describing basic use of the modules.

## Driving a module

The three essential functions of the low level API are the following:

```
int yapiInitAPI(int connection_type, char *errmsg);
int yapiUpdateDeviceList(int forceupdate, char *errmsg);
int yapiHTTPRequest(char *device, char *request, char* buffer,int buffsize,int *fullsize,
char *errmsg);
```

The *yapiInitAPI* function initializes the API and must be called once at the beginning of the program. For a USB type connection, the *connection_type* parameter takes value 1. The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message. This pointer can also point to *null*. The function returns a negative integer in case of error, zero otherwise.

The *yapiUpdateDeviceList* manages the inventory of connected Yoctopuce modules. It must be called at least once. To manage hot plug and detect potential newly connected modules, this function must be called at regular intervals. The *forceupdate* parameter must take value 1 to force a hardware scan. The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message. This pointer can also point to *null*. The function returns a negative integer in case of error, zero otherwise.

Finally, the *yapiHTTPRequest* function sends HTTP requests to the module REST API. The *device* parameter contains the serial number or the logical name of the module which you want to reach. The *request* parameter contains the full HTTP request (including terminal line breaks). *buffer* points to a character buffer long enough to contain the answer. *buffsize* is the size of the buffer. *fullsize* is a pointer to an integer to which will be assigned the actual size of the answer. The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message. This pointer can also point to *null*. The function returns a negative integer in case of error, zero otherwise.

The format of the requests is the same as the one described in the *VirtualHub et HTTP GET* section. All the character strings used by the API are strings made of 8-bit characters: Unicode and UTF8 are not supported.

The resutlt returned in the buffer variable respects the HTTP protocol. It therefore includes an HTTP header. This header ends with two empty lines, that is a sequence of four ASCII characters 13, 10, 13, 10.

Here is a sample program written in pascal using the *yapi.dll* DLL to read and then update the luminosity of a module.

```
// Dll functions import
function  yapiInitAPI(mode:integer;
                      errmsg : pansichar):integer;cdecl;
```

```
                      external 'yapi.dll' name 'yapiInitAPI';
function  yapiUpdateDeviceList(force:integer;errmsg : pansichar):integer;cdecl;
                      external 'yapi.dll' name 'yapiUpdateDeviceList';
function  yapiHTTPRequest(device:pansichar;url:pansichar; buffer:pansichar;
                      buffsize:integer;var fullsize:integer;
                      errmsg : pansichar):integer;cdecl;
                      external 'yapi.dll' name 'yapiHTTPRequest';

var
 errmsgBuffer  : array [0..256] of ansichar;
 dataBuffer    : array [0..1024] of ansichar;
 errmsg,data   : pansichar;
 fullsize,p    : integer;

const
  serial      = 'YGNSSMK1-12345';
  getValue = 'GET /api/module/luminosity HTTP/1.1'#13#10#13#10;
  setValue = 'GET /api/module?luminosity=100 HTTP/1.1'#13#10#13#10;

begin
  errmsg  :=  @errmsgBuffer;
  data    :=  @dataBuffer;
  // API  initialization
  if(yapiInitAPI(1,errmsg)<0) then
   begin
    writeln(errmsg);
    halt;
  end;

  // forces a device inventory
  if( yapiUpdateDeviceList(1,errmsg)<0) then
    begin
     writeln(errmsg);
     halt;
   end;

  // requests the  module luminosity
  if (yapiHTTPRequest(serial,getValue,data,sizeof(dataBuffer),fullsize,errmsg)<0) then
   begin
     writeln(errmsg);
     halt;
   end;

  // searches for the HTTP header end
  p := pos(#13#10#13#10,data);

  // displays the response minus the HTTP header
  writeln(copy(data,p+4,length(data)-p-3));

  // changes the luminosity
  if (yapiHTTPRequest(serial,setValue,data,sizeof(dataBuffer),fullsize,errmsg)<0) then
   begin
     writeln(errmsg);
     halt;
   end;

end.
```

## Module inventory

To perform an inventory of Yoctopuce modules, you need two functions from the dynamic library:

```
int yapiGetAllDevices(int *buffer,int maxsize,int *neededsize,char *errmsg);
int yapiGetDeviceInfo(int devdesc,yDeviceSt *infos, char *errmsg);
```

The *yapiGetAllDevices* function retrieves the list of all connected modules as a list of handles. *buffer* points to a 32-bit integer array which contains the returned handles. *maxsize* is the size in bytes of the buffer. To *neededsize* is assigned the necessary size to store all the handles. From this, you can deduce either the number of connected modules or that the input buffer is too small. The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message. This pointer can also point to *null*. The function returns a negative integer in case of error, zero otherwise.

The *yapiGetDeviceInfo* function retrieves the information related to a module from its handle. *devdesc* is a 32-bit integer representing the module and which was obtained through *yapiGetAllDevices*. *infos* points to a data structure in which the result is stored. This data structure has the following format:

| Name | Type | Size (bytes) | Description |
|---|---|---|---|
| vendorid | int | 4 | Yoctopuce USB ID |
| deviceid | int | 4 | Module USB ID |
| devrelease | int | 4 | Module version |
| nbinbterfaces | int | 4 | Number of USB interfaces used by the module |
| manufacturer | char[] | 20 | Yoctopuce (null terminated) |
| productname | char[] | 28 | Model (null terminated) |
| serial | char[] | 20 | Serial number (null terminated) |
| logicalname | char[] | 20 | Logical name (null terminated) |
| firmware | char[] | 22 | Firmware version (null terminated) |
| beacon | byte | 1 | Beacon state (0/1) |

The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message.

Here is a sample program written in pascal using the *yapi.dll* DLL to list the connected modules.

```
// device description structure
type yDeviceSt = packed record
   vendorid        : word;
   deviceid        : word;
   devrelease      : word;
   nbinbterfaces   : word;
   manufacturer    : array [0..19] of ansichar;
   productname     : array [0..27] of ansichar;
   serial          : array [0..19] of ansichar;
   logicalname     : array [0..19] of ansichar;
   firmware        : array [0..21] of ansichar;
   beacon          : byte;
 end;

// Dll function import
function  yapiInitAPI(mode:integer;
                      errmsg : pansichar):integer;cdecl;
                      external 'yapi.dll' name 'yapiInitAPI';

function  yapiUpdateDeviceList(force:integer;errmsg : pansichar):integer;cdecl;
                      external 'yapi.dll' name 'yapiUpdateDeviceList';

function  yapiGetAllDevices( buffer:pointer;
                             maxsize:integer;
                             var neededsize:integer;
                             errmsg : pansichar):integer; cdecl;
                             external 'yapi.dll' name 'yapiGetAllDevices';

function  apiGetDeviceInfo(d:integer; var infos:yDeviceSt;
                             errmsg : pansichar):integer;  cdecl;
                             external 'yapi.dll' name 'yapiGetDeviceInfo';


var
 errmsgBuffer  : array [0..256] of ansichar;
 dataBuffer    : array [0..127] of integer;   // max of 128 USB devices
 errmsg,data   : pansichar;
 neededsize,i  : integer;
 devinfos      : yDeviceSt;

begin
  errmsg  :=  @errmsgBuffer;

  // API  initialization
  if(yapiInitAPI(1,errmsg)<0) then
   begin
    writeln(errmsg);
    halt;
  end;
```

```
   // forces a device inventory
  if( yapiUpdateDeviceList(1,errmsg)<0) then
    begin
     writeln(errmsg);
     halt;
   end;

  // loads all device handles into dataBuffer
  if yapiGetAllDevices(@dataBuffer,sizeof(dataBuffer),neededsize,errmsg)<0 then
    begin
     writeln(errmsg);
     halt;
    end;

  // gets device info from each handle
  for i:=0 to  neededsize div sizeof(integer)-1 do
   begin
     if (apiGetDeviceInfo(dataBuffer[i], devinfos, errmsg)<0) then
       begin
         writeln(errmsg);
         halt;
       end;
     writeln(pansichar(@devinfos.serial)+' ('+pansichar(@devinfos.productname)+')');
   end;

end.
```

### VB6 and yapi.dll

Each entry point from the yapi.dll is duplicated. You will find one regular C-decl version and one Visual Basic 6 compatible version, prefixed with *vb6_*.

## 19.5. Porting the high level library

As all the sources of the Yoctopuce API are fully provided, you can very well port the whole API in the language of your choice. Note, however, that a large portion of the API source code is automatically generated.

Therefore, it is not necessary for you to port the complete API. You only need to port the *yocto_api* file and one file corresponding to a function, for example *yocto_relay*. After a little additional work, Yoctopuce is then able to generate all other files. Therefore, we highly recommend that you contact Yoctopuce support before undertaking to port the Yoctopuce library in another language. Collaborative work is advantageous to both parties.

# 20. Advanced programming

The preceding chapters have introduced, in each available language, the basic programming functions which can be used with your Yocto-GPS module. This chapter presents in a more generic manner a more advanced use of your module. Examples are provided in the language which is the most popular among Yoctopuce customers, that is C#. Nevertheless, you can find complete examples illustrating the concepts presented here in the programming libraries of each language.

To remain as concise as possible, examples provided in this chapter do not perform any error handling. Do not copy them "as is" in a production application.

## 20.1. Event programming

The methods to manage Yoctopuce modules which we presented to you in preceding chapters were polling functions, consisting in permanently asking the API if something had changed. While easy to understand, this programming technique is not the most efficient, nor the most reactive. Therefore, the Yoctopuce programming API also provides an event programming model. This technique consists in asking the API to signal by itself the important changes as soon as they are detected. Each time a key parameter is modified, the API calls a callback function which you have defined in advance.

### Detecting module arrival and departure

*Hot-plug* management is important when you work with USB modules because, sooner or later, you will have to connect or disconnect a module when your application is running. The API is designed to manage module unexpected arrival or departure in a transparent way. But your application must take this into account if it wants to avoid pretending to use a disconnected module.

Event programming is particularly useful to detect module connection/disconnection. Indeed, it is simpler to be told of new connections rather than to have to permanently list the connected modules to deduce which ones just arrived and which ones left. To be warned as soon as a module is connected, you need three pieces of code.

#### The callback

The callback is the function which is called each time a new Yoctopuce module is connected. It takes as parameter the relevant module.

```
 static void deviceArrival(YModule m)
{
  Console.WriteLine("New module  : " + m.get_serialNumber());
}
```

### Initialization

You must then tell the API that it must call the callback when a new module is connected.

```
YAPI.RegisterDeviceArrivalCallback(deviceArrival);
```

Note that if modules are already connected when the callback is registered, the callback is called for each of the already connected modules.

### Triggering callbacks

A classis issue of callback programming is that these callbacks can be triggered at any time, including at times when the main program is not ready to receive them. This can have undesired side effects, such as dead-locks and other race conditions. Therefore, in the Yoctopuce API, module arrival/departure callbacks are called only when the `UpdateDeviceList()` function is running. You only need to call `UpdateDeviceList()` at regular intervals from a timer or from a specific thread to precisely control when the calls to these callbacks happen:

```
// waiting loop managing callbacks
while (true)
{
    // module arrival / departure callback
    YAPI.UpdateDeviceList(ref errmsg);
    // non active waiting time managing other callbacks
    YAPI.Sleep(500, ref errmsg);
}
```

In a similar way, it is possible to have a callback when a module is disconnected. You can find a complete example implemented in your favorite programming language in the *Examples/Prog-EventBased* directory of the corresponding library.

Be aware that in most programming languages, callbacks must be global procedures, and not methods. If you wish for the callback to call the method of an object, define your callback as a global procedure which then calls your method.

## Detecting a modification in the value of a sensor

The Yoctopuce API also provides a callback system allowing you to be notified automatically with the value of any sensor, either when the value has changed in a significant way or periodically at a preset frequency. The code necessary to do so is rather similar to the code used to detect when a new module has been connected.

This technique is useful in particular if you want to detect very quick value changes (within a few milliseconds), as it is much more efficient than reading repeatedly the sensor value and therefore gives better performances.

### Callback invocation

To enable a better control, value change callbacks are only called when the `YAPI.Sleep()` and `YAPI.HandleEvents()` functions are running. Therefore, you must call one of these functions at a regular interval, either from a timer or from a parallel thread.

```
while (true)
{
  // inactive waiting loop allowing you to trigger
  // value change callbacks
  YAPI.Sleep(500, ref errmsg);
}
```

In programming environments where only the interface thread is allowed to interact with the user, it is often appropriate to call `YAPI.HandleEvents()` from this thread.

### The value change callback

This type of callback is called when a latitude sensor changes in a significant way. It takes as parameter the relevant function and the new value, as a character string.[1]

```
static void valueChangeCallback(YLatitude fct, string value)
{
  Console.WriteLine(fct.get_hardwareId() + "=" + value);
}
```

In most programming languages, callbacks are global procedures, not methods. If you wish for the callback to call a method of an object, define your callback as a global procedure which then calls your method. If you need to keep a reference to your object, you can store it directly in the YLatitude object using function `set_userData`. You can then retrieve it in the global callback procedure using `get_userData`.

### Setting up a value change callback

The callback is set up for a given Latitude function with the help of the `registerValueCallback` method. The following example sets up a callback for the first available Latitude function.

```
YLatitude f = YLatitude.FirstLatitude();
f.registerValueCallback(latitudeChangeCallBack)
```

Note that each module function can thus have its own distinct callback. By the way, if you like to work with value change callbacks, you will appreciate the fact that value change callbacks are not limited to sensors, but are also available for all Yoctopuce devices (for instance, you can also receive a callback any time a relay state changes).

### The timed report callback

This type of callback is automatically called at a predefined time interval. The callback frequency can be configured individually for each sensor, with frequencies going from hundred calls per seconds down to one call per hour. The callback takes as parameter the relevant function and the measured value, as an `YMeasure` object. Contrarily to the value change callback that only receives the latest value, an `YMeasure` object provides both minimal, maximal and average values since the timed report callback. Moreover, the measure includes precise timestamps, which makes it possible to use timed reports for a time-based graph even when not handled immediately.

```
static void periodicCallback(YLatitude fct, YMeasure measure)
{
  Console.WriteLine(fct.get_hardwareId() + "=" +
                    measure.get_averageValue());
}
```

### Setting up a timed report callback

The callback is set up for a given Latitude function with the help of the `registerTimedReportCallback` method. The callback will only be invoked once a callback frequency as been set using `set_reportFrequency` (which defaults to timed report callback turned off). The frequency is specified as a string (same as for the data logger), by specifying the number of calls per second (/s), per minute (/m) or per hour (/h). The maximal frequency is 100 times per second (i.e. "100/s"), and the minimal frequency is 1 time per hour (i.e. "1/h"). When the frequency is higher than or equal to 1/s, the measure represents an instant value. When the frequency is below, the measure will include distinct minimal, maximal and average values based on a sampling performed automatically by the device.

The following example sets up a timed report callback 4 times per minute for t he first available Latitude function.

```
YLatitude f = YLatitude.FirstLatitude();
f.set_reportFrequency("4/m");
```

[1] The value passed as parameter is the same as the value returned by the `get_advertisedValue()` method.

```
f.registerTimedReportCallback(periodicCallback);
```

As for value change callbacks, each module function can thus have its own distinct timed report callback.

**Generic callback functions**
It is sometimes desirable to use the same callback function for various types of sensors (e.g. for a generic sensor graphing application). This is possible by defining the callback for an object of class `YSensor` rather than `YLatitude`. Thus, the same callback function will be usable with any subclass of `YSensor` (and in particular with `YLatitude`). With the callback function, you can use the method `get_unt()` to get the physical unit of the sensor, if you need to display it.

**A complete example**
You can find a complete example implemented in your favorite programming language in the *Examples/Prog-EventBased* directory of the corresponding library.

# 20.2. The data logger

Your Yocto-GPS is equipped with a data logger able to store non-stop the measures performed by the module. The maximal frequency is 100 times per second (i.e. "100/s"), and the minimal frequency is 1 time per hour (i.e. "1/h"). When the frequency is higher than or equal to 1/s, the measure represents an instant value. When the frequency is below, the measure will include distinct minimal, maximal and average values based on a sampling performed automatically by the device.

Note that is useless and counter-productive to set a recording frequency higher than the native sampling frequency of the recorded sensor.

The data logger flash memory can store about 500'000 instant measures, or 125'000 averaged measures. When the memory is about to be saturated, the oldest measures are automatically erased.

Make sure not to leave the data logger running at high speed unless really needed: the flash memory can only stand a limited number of erase cycles (typically 100'000 cycles). When running at full speed, the datalogger can burn more than 100 cycles per day ! Also be aware that it is useless to record measures at a frequency higher than the refresh frequency of the physical sensor itself.

## Starting/stopping the datalogger

The data logger can be started with the `set_recording()` method.

```
YDataLogger l = YDataLogger.FirstDataLogger();
l.set_recording(YDataLogger.RECORDING_ON);
```

It is possible to make the data recording start automatically as soon as the module is powered on.

```
YDataLogger l = YDataLogger.FirstDataLogger();
l.set_autoStart(YDataLogger.AUTOSTART_ON);
l.get_module().saveToFlash();  // do not forget to save the setting
```

Note: Yoctopuce modules do not need an active USB connection to work: they start working as soon as they are powered on. The Yocto-GPS can store data without necessarily being connected to a computer: you only need to activate the automatic start of the data logger and to power on the module with a simple USB charger.

## Erasing the memory

The memory of the data logger can be erased with the `forgetAllDataStreams()` function. Be aware that erasing cannot be undone.

```
YDataLogger l = YDataLogger.FirstDataLogger();
l.forgetAllDataStreams();
```

## Choosing the logging frequency

The logging frequency can be set up individually for each sensor, using the method `set_logFrequency()`. The frequency is specified as a string (same as for timed report callbacks), by specifying the number of calls per second (/s), per minute (/m) or per hour (/h). The default value is "1/s".

The following example configures the logging frequency at 15 measures per minute for the first sensor found, whatever its type:

```
YSensor sensor = YSensor.FirstSensor();
sensor.set_logFrequency("15/m");
```

To avoid wasting flash memory, it is possible to disable logging for specified functions. In order to do so, simply use the value "OFF":

```
sensor.set_logFrequency("OFF");
```

**Limitation**: The Yocto-GPS cannot use a different frequency for timed-report callbacks and for recording data into the datalogger. You can disable either of them individually, but if you enable both timed-report callbacks and logging for a given function, the two will work at the same frequency.

## Retrieving the data

To load recorded measures from the Yocto-GPS flash memory, you must call the `get_recordedData()` method of the desired sensor, and specify the time interval for which you want to retrieve measures. The time interval is given by the start and stop UNIX timestamp. You can also specify 0 if you don't want any start or stop limit.

The `get_recordedData()` method does not return directly am array of measured values, since in some cases it would cause a huge load that could affect the responsiveness of the application. Instead, this function will return an `YDataSet` object that can be used to retrieve immediately an overview of the measured data (summary), and then to load progressively the details when desired.

Here are the main methods used to retrieve recorded measures:

1. **dataset = sensor.get_recordedData(0,0)**: select the desired time interval
2. **dataset.loadMore()**: load data from the device, progressively
3. **dataset.get_summary()**: get a single measure summarizing the full time interval
4. **dataset.get_preview()**: get an array of measures representing a condensed version of the whole set of measures on the selected time interval (reduced by a factor of approx. 200)
5. **dataset.get_measures()**: get an array with all detailled measures (that grows while `loadMore` is being called repeteadly)

Measures are instances of `YMeasure` [2]. They store simultaneously the minimal, average and maximal value at a given time, that you can retrieve using methods **get_minValue()**, **get_averageValue()** and **get_maxValue()** respectively. Here is a small example that uses the functions above:

```
// We will retrieve all measures, without time limit
YDataSet dataset = sensor.get_recordedData(0, 0);

// First call to loadMore() loads the summary/preview
dataset.loadMore();
YMeasure summary = dataset.get_summary();
string timeFmt = "dd MMM yyyy hh:mm:ss,fff";
string logFmt = "from {0} to {1} : average={2:0.00}{3}";
Console.WriteLine(String.Format(logFmt,
    summary.get_startTimeUTC_asDateTime().ToString(timeFmt),
    summary.get_endTimeUTC_asDateTime().ToString(timeFmt),
    summary.get_averageValue(), sensor.get_unit()));
```

[2] The `YMeasure` objects used by the data logger are exactly the same kind as those passed as argument to the timed report callbacks.

```
// Next calls to loadMore() will retrieve measures
Console.WriteLine("loading details");
int progress;
do {
    Console.Write(".");
    progress = dataset.loadMore();
} while(progress < 100);

// All measures have now been loaded
List<YMeasure> details = dataset.get_measures();
foreach (YMeasure m in details) {
    Console.WriteLine(String.Format(logFmt,
        m.get_startTimeUTC_asDateTime().ToString(timeFmt),
        m.get_endTimeUTC_asDateTime().ToString(timeFmt),
        m.get_averageValue(), sensor.get_unit()));
}
```

You will find a complete example demonstrating how to retrieve data from the logger for each programming language directly in the Yoctopuce library. The example can be found in directory *Examples/Prog-DataLogger*.

### Timestamp

As the Yocto-GPS does not have a battery, it cannot guess alone the current time when powered on. Nevertheless, the Yocto-GPS will automatically try to adjust its real-time reference using the host to which it is connected, in order to properly attach a timestamp to each measure in the datalogger:

- When the Yocto-GPS is connected to a computer running either the VirtualHub or any application using the Yoctopuce library, it will automatically receive the time from this computer.
- When the Yocto-GPS is connected to a YoctoHub-Ethernet, it will get the time that the YoctoHub has obtained from the network (using a server from `pool.ntp.org`)
- When the Yocto-GPS is connected to a YoctoHub-Wireless, it will get the time provided by the YoctoHub based on its internal battery-powered real-time clock, which was itself configured either from the network or from a computer
- When the Yocto-GPS is connected to an Android mobile device, it will get the time from the mobile device as long as an app using the Yoctopuce library is launched.

When none of these conditions applies (for instance if the module is simply connected to an USB charger), the Yocto-GPS will do its best effort to attach a reasonable timestamp to the measures, using the timestamp found on the latest recorded measures. It is therefore possible to "preset to the real time" an autonomous Yocto-GPS by connecting it to an Android mobile phone, starting the data logger, then connecting the device alone on an USB charger. Nevertheless, be aware that without external time source, the internal clock of the Yocto-GPS might be be subject to a clock skew (theoretically up to 2%).

## 20.3. Sensor calibration

Your Yocto-GPS module is equipped with a digital sensor calibrated at the factory. The values it returns are supposed to be reasonably correct in most cases. There are, however, situations where external conditions can impact the measures.

The Yoctopuce API provides the mean to re-caliber the values measured by your Yocto-GPS. You are not going to modify the hardware settings of the module, but rather to transform afterwards the measures taken by the sensor. This transformation is controlled by parameters stored in the flash memory of the module, making it specific for each module. This re-calibration is therefore a fully software matter and remains perfectly reversible.

Before deciding to re-calibrate your Yocto-GPS module, make sure you have well understood the phenomena which impact the measures of your module, and that the differences between true values and measured values do not result from a incorrect use or an inadequate location of the module.

The Yoctopuce modules support two types of calibration. On the one hand, a linear interpolation based on 1 to 5 reference points, which can be performed directly inside the Yocto-GPS. On the other hand, the API supports an external arbitrary calibration, implemented with callbacks.

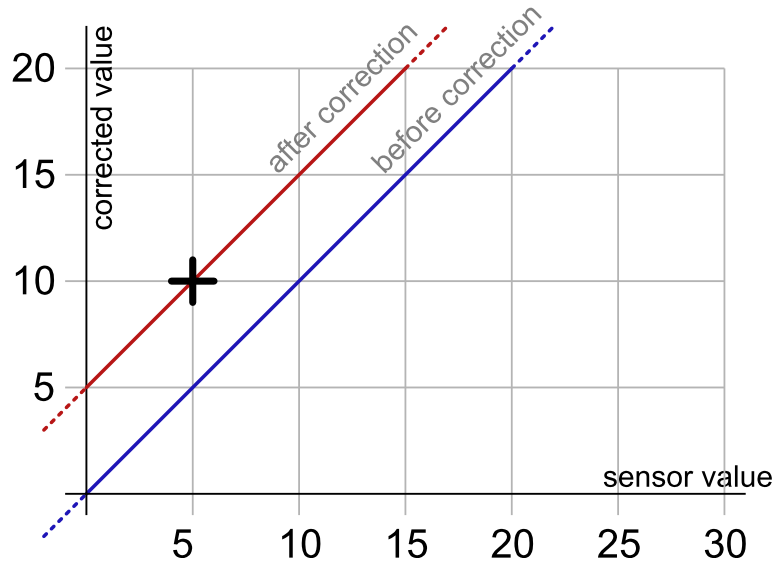
## 1 to 5 point linear interpolation

These transformations are performed directly inside the Yocto-GPS which means that you only have to store the calibration points in the module flash memory, and all the correction computations are done in a perfectly transparent manner: The function `get_currentValue()` returns the corrected value while the function `get_currentRawValue()` keeps returning the value before the correction.

Calibration points are simply *(Raw_value, Corrected_value)* couples. Let us look at the impact of the number of calibration points on the corrections.

### 1 point correction

The 1 point correction only adds a shift to the measures. For example, if you provide the calibration point *(a, b)*, all the measured values are corrected by adding to them *b-a*, so that when the value read on the sensor is *a*, the latitude function returns *b*.



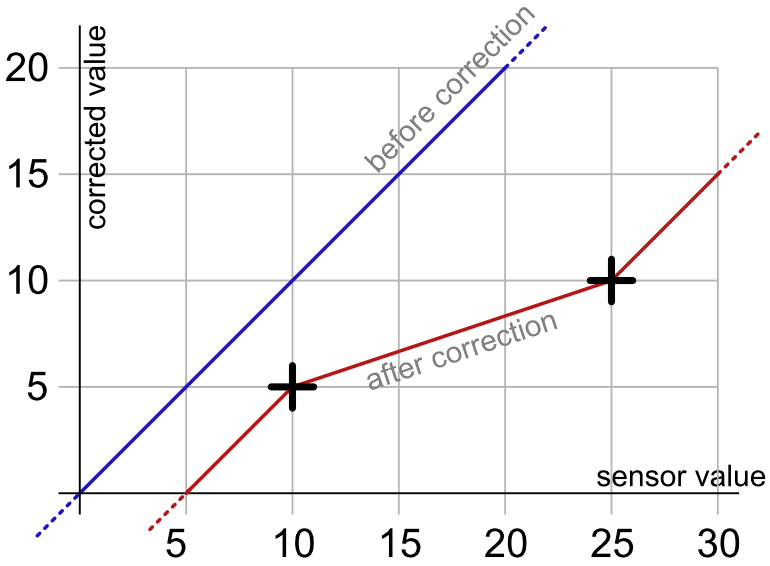

*Measure correction with 1 calibration point, here (5,10)*

The application is very simple: you only need to call the *calibrateFromPoints()* method of the function you wish to correct. The following code applies the correction illustrated on the graph above to the first latitude function found. Note the call to the *saveToFlash* method of the module hosting the function, so that the module does not forget the calibration as soon as it is disconnected.

```
Double[] ValuesBefore = {5};
Double[] ValuesAfter  = {10};
YLatitude f = YLatitude.FirstLatitude();
f.calibrateFromPoints(ValuesBefore, ValuesAfter);
f.get_module().saveToFlash();
```

### 2 point correction

2 point correction allows you to perform both a shift and a multiplication by a given factor between two points. If you provide the two points (a, b) and (c, d), the function result is multiplied *(d-b)/(c-a)* in the [a, c] range and shifted, so that when the value read by the sensor is *a* or *c*, the latitude function returns respectively *b* and *d*. Outside of the [a, c] range, the values are simply shifted, so as to preserve the continuity of the measures: an increase of 1 on the value read by the sensor induces an increase of 1 on the returned value.




*Measure correction with the two calibration points (10,5) and (25,10).*

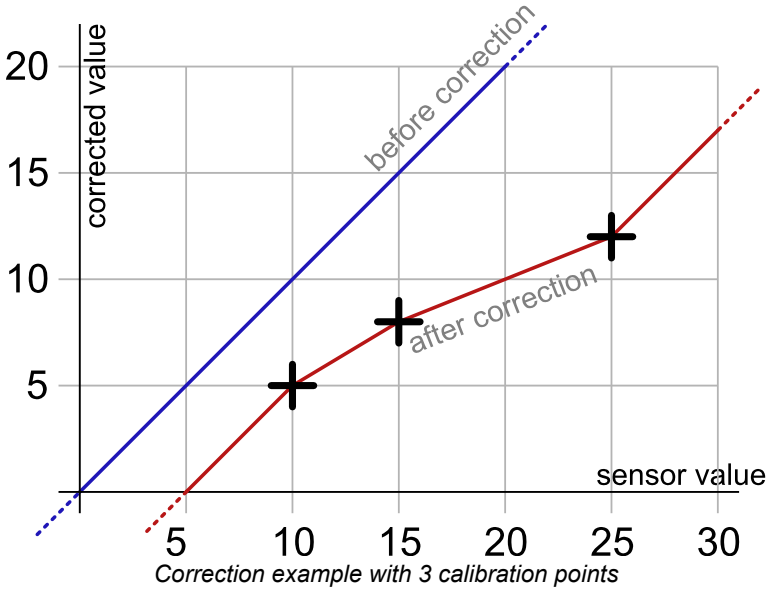
The code allowing you to program this calibration is very similar to the preceding code example.

```
Double[] ValuesBefore = {10,25};
Double[] ValuesAfter  = {5,10};
YLatitude f = YLatitude.FirstLatitude();
f.calibrateFromPoints(ValuesBefore, ValuesAfter);
f.get_module().saveToFlash();
```

Note that the values before correction must be sorted in a strictly ascending order, otherwise they are simply ignored.

**3 to 5 point correction**

3 to 5 point corrections are only a generalization of the 2 point method, allowing you to create up to 4 correction ranges for an increased precision. These ranges cannot be disjoint.




*Correction example with 3 calibration points*

**Back to normal**

To cancel the effect of a calibration on a function, call the *calibrateFromPoints()* method with two empty arrays.

```
Double[] ValuesBefore = {};
Double[] ValuesAfter  = {};
YLatitude f = YLatitude.FirstLatitude();
```

```
f.calibrateFromPoints(ValuesBefore, ValuesAfter);
f.get_module().saveToFlash();
```

You will find, in the *Examples\Prog-Calibration* directory of the Delphi, VB, and C# libraries, an application allowing you to test the effects of the 1 to 5 point calibration.

## Limitations

Due to storage and processing limitations of real values within Yoctopuce sensors, raw values and corrected values must conform to a few numeric consraints:

- Only 3 decimals are taken into account (i.e. resolution is 0.001)
- The lowest allowed value is -2'100'000
- The highest allowed value is +2'100'000

## Arbitrary interpolation

It is also possible to compute the interpolation instead of letting the module do it, in order to calculate a spline interpolation, for instance. To do so, you only need to store a callback in the API. This callback must specify the number of calibration points it is expecting.

```
public static double CustomInterpolation3Points(double rawValue, int calibType,
                  int[] parameters, double[] beforeValues, double[] afterValues)
  {  double result;
     // the value to be corrected is rawValue
     // calibration points are in beforeValues and afterValues
     result = ....    // interpolation of your choice
     return result;
   }
YAPI.RegisterCalibrationHandler(3, CustomInterpolation3Points);
```

Note that these interpolation callbacks are global, and not specific to each function. Thus, each time someone requests a value from a module which contains in its flash memory the correct number of calibration points, the corresponding callback is called to correct the value before returning it, enabling thus a perfectly transparent measure correction.

# 21. Firmware Update

There are multiples way to update the firmware of a Yoctopuce module..

## 21.1. The VirtualHub or the YoctoHub

It is possible to update the firmware directly from the web interface of the VirtualHub or the YoctoHub. The configuration panel of the module has an "upgrade" button to start a wizard that will guide you through the firmware update procedure.

In case the firmware update fails for any reason, and the module does no start anymore, simply unplug the module then plug it back while maintaining the *Yocto-button* down. The module will boot in "firmware update" mode and will appear in the VirtualHub interface below the module list.

## 21.2. The command line library

All the command line tools can update Yoctopuce modules thanks to the `downloadAndUpdate` command. The module selection mechanism works like for a traditional command. The [target] is the name of the module that you want to update. You can also use the "any" or "all" aliases, or even a name list, where the names are separated by commas, without spaces.

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

The following example updates all the Yoctopuce modules connected by USB.

```
C:\>YModule all downloadAndUpdate
ok: Yocto-PowerRelay RELAYHI1-266C8(rev=15430) is up to date.
ok: 0 / 0 hubs in 0.000000s.
ok: 0 / 0 shields in 0.000000s.
ok: 1 / 1 devices in 0.130000s 0.130000s per device.
ok: All devices are now up to date.
C:\>
```

## 21.3. The Android application Yocto-Firmware

You can update your module firmware from your Android phone or tablet with the Yocto-Firmware application. This application lists all the Yoctopuce modules connected by USB and checks if a more recent firmware is available on www.yoctopuce.com. If a more recent firmware is available, you can

update the module. The application is responsible for downloading and installing the new firmware while preserving the module parameters.

Please note: while the firmware is being updated, the module restarts several times. Android interprets a USB device reboot as a disconnection and reconnection of the USB device and asks the authorization to use the USB port again. The user must click on *OK* for the update process to end successfully.

## 21.4. Updating the firmware with the programming library

If you need to integrate firmware updates in your application, the libraries offer you an API to update your modules.[1]

### Saving and restoring parameters

The `get_allSettings()` method returns a binary buffer enabling you to save a module persistent parameters. This function is very useful to save the network configuration of a YoctoHub for example.

```
YWireless wireless = YWireless.FindWireless("reference");
YModule m = wireless.get_module();
byte[] default_config =  m.get_allSettings();
saveFile("default.bin", default_config);
...
```

You can then apply these parameters to other modules with the `set_allSettings()` method.

```
byte[] default_config = loadFile("default.bin");
YModule m = YModule.FirstModule();
while (m != null) {
  if (m.get_productName() == "YoctoHub-Wireless") {
    m.set_allSettings(default_config);
  }
  m = m.next();
}
```

### Finding the correct firmware

The first step to update a Yoctopuce module is to find which firmware you must use. The `checkFirmware(path, onlynew)` method of the YModule object does exactly this. The method checks that the firmware given as argument (`path`) is compatible with the module. If the `onlynew` parameter is set, this method checks that the firmware is more recent than the version currently used by the module. When the file is not compatible (or if the file is older than the installed version), this method returns an empty string. In the opposite, if the file is valid, the method returns a file access path.

The following piece of code checks that the `c:\tmp\METEOMK1.17328.byn` is compatible with the module stored in the *m* variable .

```
YModule m = YModule.FirstModule();
...
...
string path = "c:\\tmp\METEOMK1.17328.byn";
string newfirm = m.checkFirmware(path, false);
if (newfirm != "") {
  Console.WriteLine("firmware " + newfirm + " is compatible");
}
...
```

[1] The JavaScript, Node.js, and PHP libraries do not yet allow you to update the modules. These functions will be available in a next build.

The argument can be a directory (instead of a file). In this case, the method checks all the files of the directory recursively and returns the most recent compatible firmware. The following piece of code checks whether there is a more recent firmware in the `c:\tmp\` directory.

```
YModule m = YModule.FirstModule();
...
...
string path = "c:\\tmp";
string newfirm = m.checkFirmware(path, true);
if (newfirm != "") {
  Console.WriteLine("firmware " + newfirm + " is compatible and newer");
}
...
```

You can also give the "www.yoctopuce.com" string as argument to check whether there is a more recent published firmware on Yoctopuce's web site. In this case, the method returns the firmware URL. You can use this URL to download the firmware on your disk or use this URL when updating the firmware (see below). Obviously, this possibility works only if your machine is connected to Internet.

```
YModule m = YModule.FirstModule();
...
...
string url = m.checkFirmware("www.yoctopuce.com", true);
if (url != "") {
  Console.WriteLine("new firmware is available at " + url );
}
...
```

## Updating the firmware

A firmware update can take several minutes. That is why the update process is run as a background task and is driven by the user code thanks to the `YFirmwareUdpate` class.

To update a Yoctopuce module, you must obtain an instance of the `YFirmwareUdpate` class with the `updateFirmware` method of a YModule object. The only parameter of this method is the *path* of the firmware that you want to install. This method does not immediately start the update, but returns a `YFirmwareUdpate` object configured to update the module.

```
string newfirm = m.checkFirmware("www.yoctopuce.com", true);
.....
YFirmwareUpdate fw_update = m.updateFirmware(newfirm);
```

The `startUpdate()` method starts the update as a background task. This background task automatically takes care of

1. saving the module parameters
2. restarting the module in "update" mode
3. updating the firmware
4. starting the module with the new firmware version
5. restoring the parameters

The `get_progress()` and `get_progressMessage()` methods enable you to follow the progression of the update. `get_progress()` returns the progression as a percentage (100 = update complete). `get_progressMessage()` returns a character string describing the current operation (deleting, writing, rebooting, ...). If the `get_progress` method returns a negative value, the update process failed. In this case, the `get_progressMessage()` returns an error message.

The following piece of code starts the update and displays the progress on the standard output.

```
YFirmwareUpdate fw_update = m.updateFirmware(newfirm);
....
int status = fw_update.startUpdate();
while (status < 100 && status >= 0) {
```

```
  int newstatus = fw_update.get_progress();
  if (newstatus != status) {
    Console.WriteLine(status + "% "
      + fw_update.get_progressMessage());
  }
  YAPI.Sleep(500, ref errmsg);
  status = newstatus;
}

if (status < 0) {
  Console.WriteLine("Firmware Update failed: "
    + fw_update.get_progressMessage());
} else {
  Console.WriteLine("Firmware Updated Successfully!");
}
```

### An Android characteristic

You can update a module firmware using the Android library. However, for modules connected by USB, Android asks the user to authorize the application to access the USB port.

During firmware update, the module restarts several times. Android interprets a USB device reboot as a disconnection and a reconnection to the USB port, and prevents all USB access as long as the user has not closed the pop-up window. The use has to click on *OK* for the update process to continue correctly. **You cannot update a module connected by USB to an Android device without having the user interacting with the device.**

## 21.5. The "update" mode

If you want to erase all the parameters of a module or if your module does not start correctly anymore, you can install a firmware from the "update" mode.

To force the module to work in "update" mode, disconnect it, wait a few seconds, and reconnect it while maintaining the *Yocto-button* down. This will restart the module in "update" mode. This update mode is protected against corruptions and is always available.

In this mode, the module is not detected by the YModule objects anymore. To obtain the list of connected modules in "update" mode, you must use the `YAPI.GetAllBootLoaders()` function. This function returns a character string array with the serial numbers of the modules in "update" mode.

```
List<string> allBootLoader = YAPI.GetAllBootLoaders();
```

The update process is identical to the standard case (see the preceding section), but you must manually instantiate the `YFirmwareUpdate` object instead of calling `module.updateFirmware()`. The constructor takes as argument three parameters: the module serial number, the path of the firmware to be installed, and a byte array with the parameters to be restored at the end of the update (or `null` to restore default parameters).

```
YFirmwareUpdateupdate fw_update;
fw_update = new YFirmwareUpdate(allBootLoader[0], newfirm, null);
int status = fw_update.startUpdate();
.....
```

# 22. High-level API Reference

This chapter summarizes the high-level API functions to drive your Yocto-GPS. Syntax and exact type names may vary from one language to another, but, unless otherwise stated, all the functions are available in every language. For detailed information regarding the types of arguments and return values for a given language, refer to the definition file for this language (`yocto_api.*` as well as the other `yocto_*` files that define the function interfaces).

For languages which support exceptions, all of these functions throw exceptions in case of error by default, rather than returning the documented error value for each function. This is by design, to facilitate debugging. It is however possible to disable the use of exceptions using the `yDisableExceptions()` function, in case you prefer to work with functions that return error values.

This chapter does not repeat the programming concepts described earlier, in order to stay as concise as possible. In case of doubt, do not hesitate to go back to the chapter describing in details all configurable attributes.

# 22.1. Class YAPI

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:

| | |
|---|---|
| dnp | import YoctoProxyAPI.YAPIProxy |
| java | import com.yoctopuce.YoctoAPI.YAPI; |
| js | <script type='text/javascript' src='yocto_api.js'></script> |
| cpp | #include "yocto_api.h" |
| m | #import "yocto_api.h" |
| pas | uses yocto_api; |
| vb | yocto_api.vb |
| cs | yocto_api.cs |
| uwp | import com.yoctopuce.YoctoAPI.YModule; |
| py | from yocto_api import * |
| php | require_once('yocto_api.php'); |
| es | in HTML: <script src="../../lib/yocto_api.js"></script><br>in node.js: require('yoctolib-es2017/yocto_api.js'); |
| vi | YModule.vi |

| Global functions |
|---|
| **YAPI.CheckLogicalName**(**name**)<br>Checks if a given string is valid as logical name for a module or a function. |
| **YAPI.ClearHTTPCallbackCacheDir**(**bool_removeFiles**)<br>Disables the HTTP callback cache. |
| **YAPI.DisableExceptions**()<br>Disables the use of exceptions to report runtime errors. |
| **YAPI.EnableExceptions**()<br>Re-enables the use of exceptions for runtime error handling. |
| **YAPI.EnableUSBHost**(**osContext**)<br>This function is used only on Android. |
| **YAPI.FreeAPI**()<br>Frees dynamically allocated memory blocks used by the Yoctopuce library. |
| **YAPI.GetAPIVersion**()<br>Returns the version identifier for the Yoctopuce library in use. |
| **YAPI.GetCacheValidity**()<br>Returns the validity period of the data loaded by the library. |
| **YAPI.GetDeviceListValidity**()<br>Returns the delay between each forced enumeration of the used YoctoHubs. |
| **YAPI.GetDllArchitecture**()<br>Returns the system architecture for the Yoctopuce communication library in use. |
| **YAPI.GetDllPath**() |

Returns the paths of the DLLs for the Yoctopuce library in use.

**YAPI.GetLog**(**lastLogLine**)

Retrieves Yoctopuce low-level library diagnostic logs.

**YAPI.GetNetworkTimeout**()

Returns the network connection delay for `yRegisterHub()` and `yUpdateDeviceList()`.

**YAPI.GetTickCount**()

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

**YAPI.HandleEvents**(**errmsg**)

Maintains the device-to-library communication channel.

**YAPI.InitAPI**(**mode**, **errmsg**)

Initializes the Yoctopuce programming library explicitly.

**YAPI.PreregisterHub**(**url**, **errmsg**)

Fault-tolerant alternative to `yRegisterHub()`.

**YAPI.RegisterDeviceArrivalCallback**(**arrivalCallback**)

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

**YAPI.RegisterDeviceRemovalCallback**(**removalCallback**)

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

**YAPI.RegisterHub**(**url**, **errmsg**)

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

**YAPI.RegisterHubDiscoveryCallback**(**hubDiscoveryCallback**)

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

**YAPI.RegisterHubWebsocketCallback**(**ws**, **errmsg**, **authpwd**)

Variant to `yRegisterHub()` used to initialize Yoctopuce API on an existing Websocket session, as happens for incoming WebSocket callbacks.

**YAPI.RegisterLogFunction**(**logfun**)

Registers a log callback function.

**YAPI.SelectArchitecture**(**arch**)

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

**YAPI.SetCacheValidity**(**cacheValidityMs**)

Change the validity period of the data loaded by the library.

**YAPI.SetDelegate**(**object**)

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

**YAPI.SetDeviceListValidity**(**deviceListValidity**)

Modifies the delay between each forced enumeration of the used YoctoHubs.

**YAPI.SetHTTPCallbackCacheDir**(**str_directory**)

Enables the HTTP callback cache.

**YAPI.SetNetworkTimeout**(**networkMsTimeout**)

Modifies the network connection delay for `yRegisterHub()` and `yUpdateDeviceList()`.

**YAPI.SetTimeout**(**callback**, **ms_timeout**, **args**)

Invoke the specified callback function after a given timeout.

**YAPI.SetUSBPacketAckMs**(**pktAckDelay**)

Enables the acknowledge of every USB packet received by the Yoctopuce library.

**YAPI.Sleep**(**ms_duration**, **errmsg**)

Pauses the execution flow for a specified duration.

**YAPI.TestHub**(**url**, **mstimeout**, **errmsg**)

Test if the hub is reachable.

**YAPI.TriggerHubDiscovery(errmsg)**

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

**YAPI.UnregisterHub(url)**

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

**YAPI.UpdateDeviceList(errmsg)**

Triggers a (re)detection of connected Yoctopuce modules.

**YAPI.UpdateDeviceList_async(callback, context)**

Triggers a (re)detection of connected Yoctopuce modules.

## YAPI.CheckLogicalName() YAPI
## YAPI.CheckLogicalName()

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

```
js      function yCheckLogicalName( name)
cpp     bool yCheckLogicalName( string name)
m       +(BOOL) CheckLogicalName :(NSString *) name
pas     boolean yCheckLogicalName( name: string): boolean
vb      function yCheckLogicalName( ByVal name As String) As Boolean
cs      static bool CheckLogicalName( string name)
java    boolean CheckLogicalName( String name)
uwp     bool CheckLogicalName( string name)
py      CheckLogicalName( name)
php     function yCheckLogicalName( $name)
es      async CheckLogicalName( name)
```

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

**Parameters :**

**name** a string containing the name to check.

**Returns :**

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

## YAPI.ClearHTTPCallbackCacheDir() YAPI

YAPI.ClearHTTPCallbackCacheDir()

Disables the HTTP callback cache.

```
php function yClearHTTPCallbackCacheDir( $bool_removeFiles)
```

This method disables the HTTP callback cache, and can additionally cleanup the cache directory.

**Parameters :**

**bool_removeFiles** True to clear the content of the cache.

**Returns :**

nothing.

## YAPI.DisableExceptions() YAPI
## YAPI.DisableExceptions()

Disables the use of exceptions to report runtime errors.

| | |
|---|---|
| js | function **yDisableExceptions**( ) |
| cpp | void **yDisableExceptions**( ) |
| m | +(void) **DisableExceptions** |
| pas | **yDisableExceptions**( ) |
| vb | procedure **yDisableExceptions**( ) |
| cs | static void **DisableExceptions**( ) |
| uwp | void **DisableExceptions**( ) |
| py | **DisableExceptions**( ) |
| php | function **yDisableExceptions**( ) |
| es | async **DisableExceptions**( ) |

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

## YAPI.EnableExceptions() YAPI
## YAPI.EnableExceptions()

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

| | |
|---|---|
| js | function **yEnableExceptions**( ) |
| cpp | void **yEnableExceptions**( ) |
| m | +(void) **EnableExceptions** |
| pas | **yEnableExceptions**( ) |
| vb | procedure **yEnableExceptions**( ) |
| cs | static void **EnableExceptions**( ) |
| uwp | void **EnableExceptions**( ) |
| py | **EnableExceptions**( ) |
| php | function **yEnableExceptions**( ) |
| es | async **EnableExceptions**( ) |

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

## YAPI.EnableUSBHost() YAPI
## YAPI.EnableUSBHost()

This function is used only on Android.

`java` void **EnableUSBHost**( Object **osContext**)

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

**Parameters :**

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

## YAPI.FreeAPI() YAPI

## YAPI.FreeAPI()

Frees dynamically allocated memory blocks used by the Yoctopuce library.

| | |
|---|---|
| js | function **yFreeAPI**( ) |
| cpp | void **yFreeAPI**( ) |
| m | +(void) **FreeAPI** |
| pas | **yFreeAPI**( ) |
| vb | procedure **yFreeAPI**( ) |
| cs | static void **FreeAPI**( ) |
| dnp | static void **FreeAPI**( ) |
| java | void **FreeAPI**( ) |
| uwp | void **FreeAPI**( ) |
| py | **FreeAPI**( ) |
| php | function **yFreeAPI**( ) |
| es | async **FreeAPI**( ) |

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

## YAPI.GetAPIVersion() YAPI

## YAPI.GetAPIVersion()

Returns the version identifier for the Yoctopuce library in use.

| | |
|---|---|
| js | function **yGetAPIVersion**( ) |
| cpp | string **yGetAPIVersion**( ) |
| m | +(NSString*) **GetAPIVersion** |
| pas | string **yGetAPIVersion**( ): string |
| vb | function **yGetAPIVersion**( ) As String |
| cs | static String **GetAPIVersion**( ) |
| dnp | static string **GetAPIVersion**( ) |
| java | static String **GetAPIVersion**( ) |
| uwp | static string **GetAPIVersion**( ) |
| py | **GetAPIVersion**( ) |
| php | function **yGetAPIVersion**( ) |
| es | async **GetAPIVersion**( ) |

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

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

**Returns :**
a character string describing the library version.

## YAPI.GetCacheValidity() YAPI
## YAPI.GetCacheValidity()

Returns the validity period of the data loaded by the library.

| | |
|---|---|
| cpp | static u64 **yGetCacheValidity**( ) |
| m | +(u64) **GetCacheValidity** |
| pas | u64 **yGetCacheValidity**( ): u64 |
| vb | function **yGetCacheValidity**( ) As Long |
| cs | ulong **GetCacheValidity**( ) |
| java | long **GetCacheValidity**( ) |
| uwp | async Task<ulong> **GetCacheValidity**( ) |
| py | **GetCacheValidity**( ) |
| php | function **yGetCacheValidity**( ) |
| es | async **GetCacheValidity**( ) |

This method returns the cache validity of all attributes module functions. Note: This function must be called after `yInitAPI` .

**Returns :**
an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

## YAPI.GetDeviceListValidity() YAPI
## YAPI.GetDeviceListValidity()

Returns the delay between each forced enumeration of the used YoctoHubs.

| | |
|---|---|
| cpp | static int **yGetDeviceListValidity**( ) |
| m | +(int) **GetDeviceListValidity** |
| pas | LongInt **yGetDeviceListValidity**( ): LongInt |
| vb | function **yGetDeviceListValidity**( ) As Integer |
| cs | int **GetDeviceListValidity**( ) |
| java | int **GetDeviceListValidity**( ) |
| uwp | async Task<int> **GetDeviceListValidity**( ) |
| py | **GetDeviceListValidity**( ) |
| php | function **yGetDeviceListValidity**( ) |
| es | async **GetDeviceListValidity**( ) |

Note: you must call this function after `yInitAPI`.

**Returns :**
the number of seconds between each enumeration.

## YAPI.GetDllArchitecture() YAPI
## YAPI.GetDllArchitecture()

Returns the system architecture for the Yoctopuce communication library in use.

dnp static string **GetDllArchitecture**( )

On Windows, the architecture can be `"Win32"` or `"Win64"`. On ARM machines, the architecture is `"Armhf32"` or `"Aarch64"`. On other Linux machines, the architecture is `"Linux32"` or `"Linux64"`. On MacOS, the architecture is `"MacOs32"` or `"MacOs64"`.

**Returns :**
a character string describing the system architecture of the low-level communication library.

## YAPI.GetDllPath() YAPI

## YAPI.GetDllPath()

Returns the paths of the DLLs for the Yoctopuce library in use.

dnp static string **GetDllPath**( )

For architectures that require multiple DLLs, in particular when using a .NET assembly DLL, the returned string takes the form `"DotNetProxy=/...; yapi=/...;"`, where the first path corresponds to the .NET assembly DLL and the second path corresponds to the low-level communication library.

**Returns :**
a character string describing the DLL path.

## YAPI.GetLog() YAPI
## YAPI.GetLog()

Retrieves Yoctopuce low-level library diagnostic logs.

dnp static string **GetLog**( string **lastLogLine**)

This method allows to progessively retrieve API logs. The interface is line-based: it must called it within a loop until the returned value is an empty string. Make sure to exit the loop when an empty string is returned, as feeding an empty string into the `lastLogLine` parameter for the next call would restart enumerating logs from the oldest message available.

**Parameters :**

**lastLogLine** On first call, provide an empty string. On subsequent calls, provide the last log line returned by `GetLog()`.

**Returns :**

a string with the log line immediately following the one given in argument, if such line exist. Returns an empty string otherwise, when completed.

## YAPI.GetNetworkTimeout() YAPI

## YAPI.GetNetworkTimeout()

Returns the network connection delay for `yRegisterHub()` and `yUpdateDeviceList()`.

| | |
|---|---|
| cpp | static int **yGetNetworkTimeout**( ) |
| m | +(int) **GetNetworkTimeout** |
| pas | LongInt **yGetNetworkTimeout**( ): LongInt |
| vb | function **yGetNetworkTimeout**( ) As Integer |
| cs | int **GetNetworkTimeout**( ) |
| dnp | static int **GetNetworkTimeout**( ) |
| java | int **GetNetworkTimeout**( ) |
| uwp | async Task<int> **GetNetworkTimeout**( ) |
| py | **GetNetworkTimeout**( ) |
| php | function **yGetNetworkTimeout**( ) |
| es | async **GetNetworkTimeout**( ) |

This delay impacts only the YoctoHubs and VirtualHub which are accessible through the network. By default, this delay is of 20000 milliseconds, but depending or you network you may want to change this delay, for example if your network infrastructure is based on a GSM connection.

**Returns :**
the network connection delay in milliseconds.

## YAPI.GetTickCount() YAPI
## YAPI.GetTickCount()

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

| | |
|---|---|
| js | function **yGetTickCount**( ) |
| cpp | u64 **yGetTickCount**( ) |
| m | +(u64) **GetTickCount** |
| pas | u64 **yGetTickCount**( ): u64 |
| vb | function **yGetTickCount**( ) As Long |
| cs | static ulong **GetTickCount**( ) |
| java | static long **GetTickCount**( ) |
| uwp | static ulong **GetTickCount**( ) |
| py | **GetTickCount**( ) |
| php | function **yGetTickCount**( ) |
| es | **GetTickCount**( ) |

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

**Returns :**
a long integer corresponding to the millisecond counter.

## YAPI.HandleEvents() YAPI
## YAPI.HandleEvents()

Maintains the device-to-library communication channel.

```
js      function yHandleEvents( errmsg)
cpp     YRETCODE yHandleEvents( string errmsg)
m       +(YRETCODE) HandleEvents :(NSError**) errmsg
pas     integer yHandleEvents( var errmsg: string): integer
vb      function yHandleEvents( ByRef errmsg As String) As YRETCODE
cs      static YRETCODE HandleEvents( ref string errmsg)
java    int HandleEvents( )
uwp     async Task<int> HandleEvents( )
py      HandleEvents( errmsg=None)
php     function yHandleEvents( &$errmsg)
es      async HandleEvents( errmsg)
```

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

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

**Parameters :**

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

**Returns :**

YAPI_SUCCESS when the call succeeds.

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

## YAPI.InitAPI() YAPI
## YAPI.InitAPI()

Initializes the Yoctopuce programming library explicitly.

```
js      function yInitAPI( mode, errmsg)
cpp     YRETCODE yInitAPI( int mode, string errmsg)
m       +(YRETCODE) InitAPI :(int) mode :(NSError**) errmsg
pas     integer yInitAPI( mode: integer, var errmsg: string): integer
vb      function yInitAPI( ByVal mode As Integer, ByRef errmsg As String) As Integer
cs      static int InitAPI( int mode, ref string errmsg)
java    int InitAPI( int mode)
uwp     async Task<int> InitAPI( int mode)
py      InitAPI( mode, errmsg=None)
php     function yInitAPI( $mode, &$errmsg)
es      async InitAPI( mode, errmsg)
```

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

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

**Parameters :**

**mode** an integer corresponding to the type of automatic device detection to use. Possible values are `Y_DETECT_NONE`, `Y_DETECT_USB`, `Y_DETECT_NET`, and `Y_DETECT_ALL`.

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

**Returns :**

`YAPI_SUCCESS` when the call succeeds.

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

## YAPI.PreregisterHub() YAPI

## YAPI.PreregisterHub()

Fault-tolerant alternative to `yRegisterHub()`.

```
js      function yPreregisterHub( url, errmsg)
cpp     YRETCODE yPreregisterHub( string url, string errmsg)
m       +(YRETCODE) PreregisterHub :(NSString *) url :(NSError**) errmsg
pas     integer yPreregisterHub( url: string, var errmsg: string): integer
vb      function yPreregisterHub( ByVal url As String,
                                  ByRef errmsg As String) As Integer
cs      static int PreregisterHub( string url, ref string errmsg)
dnp     static string PreregisterHub( string url)
java    int PreregisterHub( String url)
uwp     async Task<int> PreregisterHub( string url)
py      PreregisterHub( url, errmsg=None)
php     function yPreregisterHub( $url, &$errmsg)
es      async PreregisterHub( url, errmsg)
```

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

**Parameters :**

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

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

**Returns :**

`YAPI_SUCCESS` when the call succeeds.

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

## YAPI.RegisterDeviceArrivalCallback() YAPI

YAPI.RegisterDeviceArrivalCallback()

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

| | |
|---|---|
| js | function **yRegisterDeviceArrivalCallback**( **arrivalCallback**) |
| cpp | void **yRegisterDeviceArrivalCallback**( yDeviceUpdateCallback **arrivalCallback**) |
| m | +(void) **RegisterDeviceArrivalCallback** :(yDeviceUpdateCallback) **arrivalCallback** |
| pas | **yRegisterDeviceArrivalCallback**( **arrivalCallback**: yDeviceUpdateFunc) |
| vb | procedure **yRegisterDeviceArrivalCallback**( ByVal **arrivalCallback** As yDeviceUpdateFunc) |
| cs | static void **RegisterDeviceArrivalCallback**( yDeviceUpdateFunc **arrivalCallback**) |
| java | void **RegisterDeviceArrivalCallback**( DeviceArrivalCallback **arrivalCallback**) |
| uwp | void **RegisterDeviceArrivalCallback**( DeviceUpdateHandler **arrivalCallback**) |
| py | **RegisterDeviceArrivalCallback**( **arrivalCallback**) |
| php | function **yRegisterDeviceArrivalCallback**( **$arrivalCallback**) |
| es | async **RegisterDeviceArrivalCallback**( **arrivalCallback**) |

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

**Parameters :**

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

## YAPI.RegisterDeviceRemovalCallback() YAPI
## YAPI.RegisterDeviceRemovalCallback()

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

| | |
|---|---|
| js | function **yRegisterDeviceRemovalCallback**( **removalCallback**) |
| cpp | void **yRegisterDeviceRemovalCallback**( yDeviceUpdateCallback **removalCallback**) |
| m | +(void) **RegisterDeviceRemovalCallback** :(yDeviceUpdateCallback) **removalCallback** |
| pas | **yRegisterDeviceRemovalCallback**( **removalCallback**: yDeviceUpdateFunc) |
| vb | procedure **yRegisterDeviceRemovalCallback**( ByVal **removalCallback** As yDeviceUpdateFunc) |
| cs | static void **RegisterDeviceRemovalCallback**( yDeviceUpdateFunc **removalCallback**) |
| java | void **RegisterDeviceRemovalCallback**( DeviceRemovalCallback **removalCallback**) |
| uwp | void **RegisterDeviceRemovalCallback**( DeviceUpdateHandler **removalCallback**) |
| py | **RegisterDeviceRemovalCallback**( **removalCallback**) |
| php | function **yRegisterDeviceRemovalCallback**( **$removalCallback**) |
| es | async **RegisterDeviceRemovalCallback**( **removalCallback**) |

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

**Parameters :**

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

## YAPI.RegisterHub() YAPI

## YAPI.RegisterHub()

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

```
js      function yRegisterHub( url, errmsg)
cpp     YRETCODE yRegisterHub( string url, string errmsg)
m       +(YRETCODE) RegisterHub :(NSString *) url :(NSError**) errmsg
pas     integer yRegisterHub( url: string, var errmsg: string): integer
vb      function yRegisterHub( ByVal url As String,
                               ByRef errmsg As String) As Integer
cs      static int RegisterHub( string url, ref string errmsg)
dnp     static string RegisterHub( string url)
java    int RegisterHub( String url)
uwp     async Task<int> RegisterHub( string url)
py      RegisterHub( url, errmsg=None)
php     function yRegisterHub( $url, &$errmsg)
es      async RegisterHub( url, errmsg)
```

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

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

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

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

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

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

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

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

**Parameters :**

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

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

**Returns :**
YAPI_SUCCESS when the call succeeds.

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

## YAPI.RegisterHubDiscoveryCallback() YAPI
## YAPI.RegisterHubDiscoveryCallback()

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

| | |
|---|---|
| cpp | void **yRegisterHubDiscoveryCallback**( YHubDiscoveryCallback **hubDiscoveryCallback**) |
| m | +(void) **RegisterHubDiscoveryCallback** : (YHubDiscoveryCallback) **hubDiscoveryCallback** |
| pas | **yRegisterHubDiscoveryCallback**( **hubDiscoveryCallback**: YHubDiscoveryCallback) |
| vb | procedure **yRegisterHubDiscoveryCallback**( ByVal **hubDiscoveryCallback** As YHubDiscoveryCallback) |
| cs | static void **RegisterHubDiscoveryCallback**( YHubDiscoveryCallback **hubDiscoveryCallback**) |
| java | void **RegisterHubDiscoveryCallback**( HubDiscoveryCallback **hubDiscoveryCallback**) |
| uwp | async Task **RegisterHubDiscoveryCallback**( HubDiscoveryHandler **hubDiscoveryCallback**) |
| py | **RegisterHubDiscoveryCallback**( **hubDiscoveryCallback**) |
| es | async **RegisterHubDiscoveryCallback**( **hubDiscoveryCallback**) |

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

**Parameters :**

**hubDiscoveryCallback** a procedure taking two string parameter, the serial

## YAPI.RegisterHubWebsocketCallback() YAPI
## YAPI.RegisterHubWebsocketCallback()

Variant to `yRegisterHub()` used to initialize Yoctopuce API on an existing Websocket session, as happens for incoming WebSocket callbacks.

**Parameters :**

| | |
|---|---|
| **ws** | node WebSocket object for the incoming WebSocket callback connection |
| **errmsg** | a string passed by reference to receive any error message. |
| **authpwd** | the optional authentication password, required only authentication is configured on the calling hub. |

**Returns :**

`YAPI_SUCCESS` when the call succeeds.

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

## YAPI.RegisterLogFunction() YAPI

YAPI.RegisterLogFunction()

Registers a log callback function.

| | |
|---|---|
| cpp | void **yRegisterLogFunction**( yLogFunction **logfun**) |
| m | +(void) **RegisterLogFunction** :(yLogCallback) **logfun** |
| pas | **yRegisterLogFunction**( **logfun**: yLogFunc) |
| vb | procedure **yRegisterLogFunction**( ByVal **logfun** As yLogFunc) |
| cs | static void **RegisterLogFunction**( yLogFunc **logfun**) |
| java | void **RegisterLogFunction**( LogCallback **logfun**) |
| uwp | void **RegisterLogFunction**( LogHandler **logfun**) |
| py | **RegisterLogFunction**( **logfun**) |
| es | async **RegisterLogFunction**( **logfun**) |

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

**Parameters :**

**logfun** a procedure taking a string parameter, or `null`

## YAPI.SelectArchitecture() YAPI
## YAPI.SelectArchitecture()

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

py **SelectArchitecture**( **arch**)

By default, the Python library automatically detects the appropriate library to use. However, for Linux ARM, it not possible to reliably distinguish between a Hard Float (armhf) and a Soft Float (armel) install. For in this case, it is therefore recommended to manually select the proper architecture by calling `SelectArchitecture()` before any other call to the library.

**Parameters :**

**arch** A string containing the architecture to use. Possibles value are: `"armhf"`,`"armel"`, `"aarch64"`,`"i386"`,`"x86_64"`, `"32bit"`, `"64bit"`

**Returns :**

nothing.

On failure, throws an exception.

## YAPI.SetCacheValidity() YAPI
## YAPI.SetCacheValidity()

Change the validity period of the data loaded by the library.

| | |
|---|---|
| cpp | static void **ySetCacheValidity**( u64 **cacheValidityMs**) |
| m | +(void) **SetCacheValidity** : (u64) **cacheValidityMs** |
| pas | **ySetCacheValidity**( **cacheValidityMs**: u64) |
| vb | procedure **ySetCacheValidity**( ) |
| cs | void **SetCacheValidity**( ulong **cacheValidityMs**) |
| java | void **SetCacheValidity**( long **cacheValidityMs**) |
| uwp | async Task **SetCacheValidity**( ulong **cacheValidityMs**) |
| py | **SetCacheValidity**( **cacheValidityMs**) |
| php | function **ySetCacheValidity**( **$cacheValidityMs**) |
| es | async **SetCacheValidity**( **cacheValidityMs**) |

By default, when accessing a module, all the attributes of the module functions are automatically kept in cache for the standard duration (5 ms). This method can be used to change this standard duration, for example in order to reduce network or USB traffic. This parameter does not affect value change callbacks Note: This function must be called after `yInitAPI`.

**Parameters :**

**cacheValidityMs** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds.

## YAPI.SetDelegate() YAPI
## YAPI.SetDelegate()

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

m +(void) **SetDelegate** :(id) **object**

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

**Parameters :**

**object** an object that must follow the protocol YAPIDelegate, or nil

## YAPI.SetDeviceListValidity() YAPI
## YAPI.SetDeviceListValidity()

Modifies the delay between each forced enumeration of the used YoctoHubs.

| | |
|---|---|
| cpp | static void **ySetDeviceListValidity**( int **deviceListValidity**) |
| m | +(void) **SetDeviceListValidity** : (int) **deviceListValidity** |
| pas | **ySetDeviceListValidity**( **deviceListValidity**: LongInt) |
| vb | procedure **ySetDeviceListValidity**( ) |
| cs | void **SetDeviceListValidity**( int **deviceListValidity**) |
| java | void **SetDeviceListValidity**( int **deviceListValidity**) |
| uwp | async Task **SetDeviceListValidity**( int **deviceListValidity**) |
| py | **SetDeviceListValidity**( **deviceListValidity**) |
| php | function **ySetDeviceListValidity**( **$deviceListValidity**) |
| es | async **SetDeviceListValidity**( **deviceListValidity**) |

By default, the library performs a full enumeration every 10 seconds. To reduce network traffic, you can increase this delay. It's particularly useful when a YoctoHub is connected to the GSM network where traffic is billed. This parameter doesn't impact modules connected by USB, nor the working of module arrival/removal callbacks. Note: you must call this function after `yInitAPI`.

**Parameters :**

**deviceListValidity** nubmer of seconds between each enumeration.

## YAPI.SetHTTPCallbackCacheDir() YAPI
## YAPI.SetHTTPCallbackCacheDir()

Enables the HTTP callback cache.

```
php function ySetHTTPCallbackCacheDir( $str_directory)
```

When enabled, this cache reduces the quantity of data sent to the PHP script by 50% to 70%. To enable this cache, the method `ySetHTTPCallbackCacheDir()` must be called before any call to `yRegisterHub()`. This method takes in parameter the path of the directory used for saving data between each callback. This folder must exist and the PHP script needs to have write access to it. It is recommended to use a folder that is not published on the Web server since the library will save some data of Yoctopuce devices into this folder.

Note: This feature is supported by YoctoHub and VirtualHub since version 27750.

**Parameters :**

**str_directory** the path of the folder that will be used as cache.

**Returns :**

nothing.

On failure, throws an exception.

## YAPI.SetNetworkTimeout() YAPI
## YAPI.SetNetworkTimeout()

Modifies the network connection delay for `yRegisterHub()` and `yUpdateDeviceList()`.

| | |
|---|---|
| cpp | static void **ySetNetworkTimeout**( int **networkMsTimeout**) |
| m | +(void) **SetNetworkTimeout** : (int) **networkMsTimeout** |
| pas | **ySetNetworkTimeout**( **networkMsTimeout**: LongInt) |
| vb | procedure **ySetNetworkTimeout**( ) |
| cs | void **SetNetworkTimeout**( int **networkMsTimeout**) |
| dnp | static void **SetNetworkTimeout**( int **networkMsTimeout**) |
| java | void **SetNetworkTimeout**( int **networkMsTimeout**) |
| uwp | async Task **SetNetworkTimeout**( int **networkMsTimeout**) |
| py | **SetNetworkTimeout**( **networkMsTimeout**) |
| php | function **ySetNetworkTimeout**( **$networkMsTimeout**) |
| es | async **SetNetworkTimeout**( **networkMsTimeout**) |

This delay impacts only the YoctoHubs and VirtualHub which are accessible through the network. By default, this delay is of 20000 milliseconds, but depending or you network you may want to change this delay, gor example if your network infrastructure is based on a GSM connection.

**Parameters :**

**networkMsTimeout** the network connection delay in milliseconds.

## YAPI.SetTimeout() YAPI

## YAPI.SetTimeout()

Invoke the specified callback function after a given timeout.

```
js  function ySetTimeout( callback, ms_timeout, args)
es  SetTimeout( callback, ms_timeout, args)
```

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

**Parameters :**

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

**Returns :**

`YAPI_SUCCESS` when the call succeeds.

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

## YAPI.SetUSBPacketAckMs() YAPI

## YAPI.SetUSBPacketAckMs()

Enables the acknowledge of every USB packet received by the Yoctopuce library.

java void **SetUSBPacketAckMs**( int **pktAckDelay**)

This function allows the library to run on Android phones that tend to loose USB packets. By default, this feature is disabled because it doubles the number of packets sent and slows down the API considerably. Therefore, the acknowledge of incoming USB packets should only be enabled on phones or tablets that loose USB packets. A delay of 50 milliseconds is generally enough. In case of doubt, contact Yoctopuce support. To disable USB packets acknowledge, call this function with the value 0. Note: this feature is only available on Android.

**Parameters :**

**pktAckDelay** then number of milliseconds before the module

## YAPI.Sleep() YAPI

## YAPI.Sleep()

Pauses the execution flow for a specified duration.

```
js    function ySleep( ms_duration, errmsg)
cpp   YRETCODE ySleep( unsigned ms_duration, string errmsg)
m     +(YRETCODE) Sleep :(unsigned) ms_duration :(NSError **) errmsg
pas   integer ySleep( ms_duration: integer, var errmsg: string): integer
vb    function ySleep( ByVal ms_duration As Integer,
                       ByRef errmsg As String) As Integer
cs    static int Sleep( int ms_duration, ref string errmsg)
java  int Sleep( long ms_duration)
uwp   async Task<int> Sleep( ulong ms_duration)
py    Sleep( ms_duration, errmsg=None)
php   function ySleep( $ms_duration, &$errmsg)
es    async Sleep( ms_duration, errmsg)
```

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

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

**Parameters :**

**ms_duration** an integer corresponding to the duration of the pause, in milliseconds.

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

**Returns :**

`YAPI_SUCCESS` when the call succeeds.

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

## YAPI.TestHub() YAPI
## YAPI.TestHub()

Test if the hub is reachable.

cpp
```
YRETCODE yTestHub( string url, int mstimeout, string errmsg)
```
m
```
+(YRETCODE) TestHub : (NSString*) url
                    : (int) mstimeout
                    : (NSError**) errmsg
```
pas
```
integer yTestHub( url: string,
                  mstimeout: integer,
                  var errmsg: string): integer
```
vb
```
function yTestHub( ByVal url As String,
                   ByVal mstimeout As Integer,
                   ByRef errmsg As String) As Integer
```
cs
```
static int TestHub( string url, int mstimeout, ref string errmsg)
```
dnp
```
static string TestHub( string url, int mstimeout)
```
java
```
int TestHub( String url, int mstimeout)
```
uwp
```
async Task<int> TestHub( string url, uint mstimeout)
```
py
```
TestHub( url, mstimeout, errmsg=None)
```
php
```
function yTestHub( $url, $mstimeout, &$errmsg)
```
es
```
async TestHub( url, mstimeout, errmsg)
```

This method do not register the hub, it only test if the hub is usable. The url parameter follow the same convention as the `yRegisterHub` method. This method is useful to verify the authentication parameters for a hub. It is possible to force this method to return after mstimeout milliseconds.

**Parameters :**

| | |
|---|---|
| **url** | a string containing either **"usb"**,**"callback"** or the root URL of the hub to monitor |
| **mstimeout** | the number of millisecond available to test the connection. |
| **errmsg** | a string passed by reference to receive any error message. |

**Returns :**

`YAPI_SUCCESS` when the call succeeds.

On failure returns a negative error code.

## YAPI.TriggerHubDiscovery() YAPI
## YAPI.TriggerHubDiscovery()

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

| | |
|---|---|
| cpp | YRETCODE **yTriggerHubDiscovery**( string **errmsg**) |
| m | +(YRETCODE) **TriggerHubDiscovery** : (NSError**) **errmsg** |
| pas | integer **yTriggerHubDiscovery**( var **errmsg**: string): integer |
| vb | function **yTriggerHubDiscovery**( ByRef **errmsg** As String) As Integer |
| cs | static int **TriggerHubDiscovery**( ref string **errmsg**) |
| java | int **TriggerHubDiscovery**( ) |
| uwp | Task<int> **TriggerHubDiscovery**( ) |
| py | **TriggerHubDiscovery**( **errmsg**=None) |
| es | async **TriggerHubDiscovery**( **errmsg**) |

**Parameters :**

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

**Returns :**

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

## YAPI.UnregisterHub() YAPI

## YAPI.UnregisterHub()

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

| | |
|---|---|
| js | function **yUnregisterHub**( **url**) |
| cpp | void **yUnregisterHub**( string **url**) |
| m | +(void) **UnregisterHub** :(NSString *) **url** |
| pas | **yUnregisterHub**( **url**: string) |
| vb | procedure **yUnregisterHub**( ByVal **url** As String) |
| cs | static void **UnregisterHub**( string **url**) |
| java | void **UnregisterHub**( String **url**) |
| uwp | async Task **UnregisterHub**( string **url**) |
| py | **UnregisterHub**( **url**) |
| php | function **yUnregisterHub**( **$url**) |
| es | async **UnregisterHub**( **url**) |

**Parameters :**

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

## YAPI.UpdateDeviceList() YAPI
## YAPI.UpdateDeviceList()

Triggers a (re)detection of connected Yoctopuce modules.

```
js      function yUpdateDeviceList( errmsg)
cpp     YRETCODE yUpdateDeviceList( string errmsg)
m       +(YRETCODE) UpdateDeviceList :(NSError**) errmsg
pas     integer yUpdateDeviceList( var errmsg: string): integer
vb      function yUpdateDeviceList( ByRef errmsg As String) As YRETCODE
cs      static YRETCODE UpdateDeviceList( ref string errmsg)
java    int UpdateDeviceList( )
uwp     async Task<int> UpdateDeviceList( )
py      UpdateDeviceList( errmsg=None)
php     function yUpdateDeviceList( &$errmsg)
es      async UpdateDeviceList( errmsg)
```

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

This function can be called as frequently as desired to refresh the device list and to make the application aware of hot-plug events. However, since device detection is quite a heavy process, UpdateDeviceList shouldn't be called more than once every two seconds.

**Parameters :**

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

**Returns :**

`YAPI_SUCCESS` when the call succeeds.

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

## YAPI.UpdateDeviceList_async() YAPI
## YAPI.UpdateDeviceList_async()

Triggers a (re)detection of connected Yoctopuce modules.

```
js  function yUpdateDeviceList_async( callback, context)
```

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

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

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

**Parameters :**

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

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

**Returns :**

nothing : the result is provided to the callback.

# 22.2. Class YModule

Global parameters control interface for all Yoctopuce devices

The `YModule` class can be used with all Yoctopuce USB devices. 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> |
| `cpp` | #include "yocto_api.h" |
| `m` | #import "yocto_api.h" |
| `pas` | uses yocto_api; |
| `vb` | yocto_api.vb |
| `cs` | yocto_api.cs |
| `dnp` | import YoctoProxyAPI.YModuleProxy |
| `java` | import com.yoctopuce.YoctoAPI.YModule; |
| `uwp` | import com.yoctopuce.YoctoAPI.YModule; |
| `py` | from yocto_api import * |
| `php` | require_once('yocto_api.php'); |
| `es` | in HTML: <script src="../../lib/yocto_api.js"></script><br>in node.js: require('yoctolib-es2017/yocto_api.js'); |
| `vi` | YModule.vi |

| Global functions |
|---|
| **YModule.FindModule**(**func**)<br>Allows you to find a module from its serial number or from its logical name. |
| **YModule.FindModuleInContext**(**yctx**, **func**)<br>Retrieves a module for a given identifier in a YAPI context. |
| **YModule.FirstModule**()<br>Starts the enumeration of modules currently accessible. |
| **`YModule` properties** |
| **module→Beacon** *[writable]*<br>State of the localization beacon. |
| **module→FunctionId** *[read-only]*<br>Retrieves the hardware identifier of the *n*th function on the module. |
| **module→HardwareId** *[read-only]*<br>Unique hardware identifier of the module. |
| **module→IsOnline** *[read-only]*<br>Checks if the module is currently reachable. |
| **module→LogicalName** *[writable]*<br>Logical name of the module. |
| **module→Luminosity** *[writable]*<br>Luminosity of the module informative LEDs (from 0 to 100). |
| **module→ProductId** *[read-only]*<br>USB device identifier of the module. |
| **module→ProductName** *[read-only]*<br>Commercial name of the module, as set by the factory. |

**module→ProductRelease** *[read-only]*

Release number of the module hardware, preprogrammed at the factory.

**module→SerialNumber** *[read-only]*

Serial number of the module, as set by the factory.

## YModule methods

**module→checkFirmware(path, onlynew)**

Tests whether the byn file is valid for this module.

**module→clearCache()**

Invalidates the cache.

**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→functionBaseType(functionIndex)**

Retrieves the base type of the *n*th function on the module.

**module→functionCount()**

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

**module→functionId(functionIndex)**

Retrieves the hardware identifier of the *n*th function on the module.

**module→functionName(functionIndex)**

Retrieves the logical name of the *n*th function on the module.

**module→functionType(functionIndex)**

Retrieves the type 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 settings and uploaded files 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_functionIds(funType)**

Retrieve all hardware identifier that match the type passed in argument.

**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_parentHub**()
Returns the serial number of the YoctoHub on which this module is connected.

**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 release number of the module hardware, preprogrammed at the factory.

**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_subDevices**()
Returns a list of all the modules that are plugged into the current module.

**module→get_upTime**()
Returns the number of milliseconds spent since the module was powered on.

**module→get_url**()
Returns the URL used to access the module.

**module→get_usbCurrent**()
Returns the current consumed by the module on the USB bus, in milli-amps.

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

**module→get_userVar**()
Returns the value previously stored in this attribute.

**module→hasFunction**(**funcId**)
Tests if the device includes a specific function.

**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→log**(**text**)
Adds a text message to the device logs.

**module→nextModule**()
Continues the module enumeration started using `yFirstModule()`.

**module→reboot**(**secBeforeReboot**)
Schedules a simple module reboot after the given number of seconds.

**module→registerBeaconCallback**(**callback**)

Register a callback function, to be called when the localization beacon of the module has been changed.

**module→registerConfigChangeCallback(callback)**

Register a callback function, to be called when a persistent settings in a device configuration has been changed (e.g.

**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)**

Restores all the settings of the device.

**module→set_allSettingsAndFiles(settings)**

Restores all the settings and uploaded files to 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)**

Stores a 32 bit value in the device RAM.

**module→triggerConfigChangeCallback()**

Triggers a configuration change callback, to check if they are supported or not.

**module→triggerFirmwareUpdate(secBeforeReboot)**

Schedules a module reboot into special firmware update mode.

**module→updateFirmware(path)**

Prepares a firmware update of the module.

**module→updateFirmwareEx(path, force)**

Prepares a firmware update 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.

## YModule.FindModule() YModule
## YModule.FindModule()

Allows you to find a module from its serial number or from its logical name.

```
js      function yFindModule( func)
cpp     YModule* yFindModule( string func)
m       +(YModule*) FindModule : (NSString*) func
pas     TYModule yFindModule( func: string): TYModule
vb      function yFindModule( ByVal func As String) As YModule
cs      static YModule FindModule( string func)
dnp     static YModuleProxy FindModule( string func)
java    static YModule FindModule( String func)
uwp     static YModule FindModule( string func)
py      FindModule( func)
php     function yFindModule( $func)
es      static FindModule( func)
```

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

If a call to this object's is_online() method returns FALSE although you are certain that the device is plugged, make sure that you did call registerHub() at application initialization time.

**Parameters :**

**func** a string containing either the serial number or the logical name of the desired module

**Returns :**

a `YModule` object allowing you to drive the module or get additional information on the module.

## YModule.FindModuleInContext() YModule YModule.FindModuleInContext()

Retrieves a module for a given identifier in a YAPI context.

| | |
|---|---|
| java | static YModule **FindModuleInContext**( YAPIContext **yctx**, String **func**) |
| uwp | static YModule **FindModuleInContext**( YAPIContext **yctx**, string **func**) |
| es | static **FindModuleInContext**( **yctx**, **func**) |

The identifier can be specified using several formats:

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

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

**Parameters :**

**yctx** a YAPI context

**func** a string that uniquely characterizes the module, for instance `MyDevice.module`.

**Returns :**

a `YModule` object allowing you to drive the module.

## YModule.FirstModule() YModule
## YModule.FirstModule()

Starts the enumeration of modules currently accessible.

| | |
|---|---|
| js | function **yFirstModule**( ) |
| cpp | YModule * **yFirstModule**( ) |
| m | +(YModule*) **FirstModule** |
| pas | TYModule **yFirstModule**( ): TYModule |
| vb | function **yFirstModule**( ) As YModule |
| cs | static YModule **FirstModule**( ) |
| java | static YModule **FirstModule**( ) |
| uwp | static YModule **FirstModule**( ) |
| py | **FirstModule**( ) |
| php | function **yFirstModule**( ) |
| es | static **FirstModule**( ) |

Use the method `YModule.nextModule()` to iterate on the next modules.

**Returns :**

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

## module→Beacon YModule

State of the localization beacon.

dnp int **Beacon**

**Possible values:**

| | |
|---|---|
| Y_BEACON_INVALID | = 0 |
| Y_BEACON_OFF | = 1 |
| Y_BEACON_ON | = 2 |

**Writable**. Turns on or off the module localization beacon.

## module→FunctionId — YModule

Retrieves the hardware identifier of the *n*th function on the module.

dnp string **FunctionId**

@param functionIndex : the index of the function for which the information is desired, starting at 0 for the first function.

## module→HardwareId — YModule

Unique hardware identifier of the module.

dnp string **HardwareId**

The unique hardware identifier is made of the device serial number followed by string ".module".

## module→IsOnline YModule

Checks if the module is currently reachable.

dnp bool **IsOnline**

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

## module→LogicalName YModule

Logical name of the module.

dnp string **LogicalName**

**Writable**. 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.

## module→Luminosity YModule

Luminosity of the module informative LEDs (from 0 to 100).

dnp int **Luminosity**

**Writable**. Changes the luminosity of the module informative leds. The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

## module→ProductId — YModule

USB device identifier of the module.

dnp int **ProductId**

## module→ProductName — YModule

Commercial name of the module, as set by the factory.

dnp string **ProductName**

## module→ProductRelease YModule

Release number of the module hardware, preprogrammed at the factory.

dnp int **ProductRelease**

The original hardware release returns value 1, revision B returns value 2, etc.

## module→SerialNumber — YModule

Serial number of the module, as set by the factory.

dnp string **SerialNumber**

## module→checkFirmware() YModule

Tests whether the byn file is valid for this module.

```
js      function checkFirmware( path, onlynew)
cpp     string checkFirmware( string path, bool onlynew)
m       -(NSString*) checkFirmware : (NSString*) path
                                   : (bool) onlynew
pas     string checkFirmware( path: string, onlynew: boolean): string
vb      function checkFirmware( ) As String
cs      string checkFirmware( string path, bool onlynew)
dnp     string checkFirmware( string path, bool onlynew)
java    String checkFirmware( String path, boolean onlynew)
uwp     async Task<string> checkFirmware( string path, bool onlynew)
py      checkFirmware( path, onlynew)
php     function checkFirmware( $path, $onlynew)
es      async checkFirmware( path, onlynew)
cmd     YModule target checkFirmware path onlynew
```

This method is useful to test if the module needs to be updated. It is possible to pass a directory as argument instead of a file. In this case, this method returns the path of the most recent appropriate `.byn` file. If the parameter `onlynew` is true, the function discards firmwares that are older or equal to the installed firmware.

**Parameters :**

**path** the path of a byn file or a directory that contains byn files

**onlynew** returns only files that are strictly newer

**Returns :**

the path of the byn file to use or a empty string if no byn files matches the requirement

On failure, throws an exception or returns a string that start with "error:".

## module→clearCache() YModule

Invalidates the cache.

| | |
|---|---|
| js | function **clearCache**( ) |
| cpp | void **clearCache**( ) |
| m | -(void) **clearCache** |
| pas | **clearCache**( ) |
| vb | procedure **clearCache**( ) |
| cs | void **clearCache**( ) |
| java | void **clearCache**( ) |
| py | **clearCache**( ) |
| php | function **clearCache**( ) |
| es | async **clearCache**( ) |

Invalidates the cache of the module attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

## module→describe() YModule

Returns a descriptive text that identifies the module.

```
js      function describe( )
cpp     string describe( )
m       -(NSString*) describe
pas     string describe( ): string
vb      function describe( ) As String
cs      string describe( )
java    String describe( )
py      describe( )
php     function describe( )
es      async describe( )
```

The text may include either the logical name or the serial number of the module.

**Returns :**
a string that describes the module

## module→download() YModule

Downloads the specified built-in file and returns a binary buffer with its content.

```
js      function download( pathname)
cpp     string download( string pathname)
m       -(NSMutableData*) download : (NSString*) pathname
pas     TByteArray download( pathname: string): TByteArray
vb      function download( ) As Byte
cs      byte[] download( string pathname)
dnp     byte[] download( string pathname)
java    byte[] download( String pathname)
uwp     async Task<byte[]> download( string pathname)
py      download( pathname)
php     function download( $pathname)
es      async download( pathname)
cmd     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→functionBaseType() YModule

Retrieves the base type of the *n*th function on the module.

| | |
|---|---|
| js | function **functionBaseType**( **functionIndex**) |
| cpp | string **functionBaseType**( int **functionIndex**) |
| pas | string **functionBaseType**( **functionIndex**: integer): string |
| vb | function **functionBaseType**( ByVal **functionIndex** As Integer) As String |
| cs | string **functionBaseType**( int **functionIndex**) |
| java | String **functionBaseType**( int **functionIndex**) |
| py | **functionBaseType**( **functionIndex**) |
| php | function **functionBaseType**( **$functionIndex**) |
| es | async **functionBaseType**( **functionIndex**) |

For instance, the base type of all measuring functions is "Sensor".

**Parameters :**

**functionIndex** the index of the function for which the information is desired, starting at 0 for the first function.

**Returns :**

a string corresponding to the base type of the function

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

## module→functionCount() YModule

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

```
js      function functionCount( )
cpp     int functionCount( )
m       -(int) functionCount
pas     integer functionCount( ): integer
vb      function functionCount( ) As Integer
cs      int functionCount( )
java    int functionCount( )
py      functionCount( )
php     function functionCount( )
es      async functionCount( )
```

**Returns :**
the number of functions on the module

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

## module→functionId() YModule

Retrieves the hardware identifier of the *n*th function on the module.

```
js      function functionId( functionIndex)
cpp     string functionId( int functionIndex)
m       -(NSString*) functionId : (int) functionIndex
pas     string functionId( functionIndex: integer): string
vb      function functionId( ByVal functionIndex As Integer) As String
cs      string functionId( int functionIndex)
java    String functionId( int functionIndex)
py      functionId( functionIndex)
php     function functionId( $functionIndex)
es      async functionId( functionIndex)
```

**Parameters :**

**functionIndex** the index of the function for which the information is desired, starting at 0 for the first function.

**Returns :**

a string corresponding to the unambiguous hardware identifier of the requested module function

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

## module→functionName() YModule

Retrieves the logical name of the *n*th function on the module.

```
js      function functionName( functionIndex)
cpp     string functionName( int functionIndex)
m       -(NSString*) functionName : (int) functionIndex
pas     string functionName( functionIndex: integer): string
vb      function functionName( ByVal functionIndex As Integer) As String
cs      string functionName( int functionIndex)
java    String functionName( int functionIndex)
py      functionName( functionIndex)
php     function functionName( $functionIndex)
es      async functionName( functionIndex)
```

**Parameters :**

**functionIndex** the index of the function for which the information is desired, starting at 0 for the first function.

**Returns :**

a string corresponding to the logical name of the requested module function

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

## module→functionType() YModule

Retrieves the type of the *n*th function on the module.

| | |
|---|---|
| js | function **functionType**( **functionIndex**) |
| cpp | string **functionType**( int **functionIndex**) |
| pas | string **functionType**( **functionIndex**: integer): string |
| vb | function **functionType**( ByVal **functionIndex** As Integer) As String |
| cs | string **functionType**( int **functionIndex**) |
| java | String **functionType**( int **functionIndex**) |
| py | **functionType**( **functionIndex**) |
| php | function **functionType**( **$functionIndex**) |
| es | async **functionType**( **functionIndex**) |

**Parameters :**

**functionIndex** the index of the function for which the information is desired, starting at 0 for the first function.

**Returns :**

a string corresponding to the type of the function

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

## module→functionValue() YModule

Retrieves the advertised value of the *n*th function on the module.

```
js      function functionValue( functionIndex)
cpp     string functionValue( int functionIndex)
m       -(NSString*) functionValue : (int) functionIndex
pas     string functionValue( functionIndex: integer): string
vb      function functionValue( ByVal functionIndex As Integer) As String
cs      string functionValue( int functionIndex)
java    String functionValue( int functionIndex)
py      functionValue( functionIndex)
php     function functionValue( $functionIndex)
es      async functionValue( functionIndex)
```

**Parameters :**

**functionIndex** the index of the function for which the information is desired, starting at 0 for the first function.

**Returns :**

a short string (up to 6 characters) corresponding to the advertised value of the requested module function

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

## module→get_allSettings() YModule
## module→allSettings()

Returns all the settings and uploaded files of the module.

```
js      function get_allSettings( )
cpp     string get_allSettings( )
m       -(NSMutableData*) allSettings
pas     TByteArray get_allSettings( ): TByteArray
vb      function get_allSettings( ) As Byte
cs      byte[] get_allSettings( )
dnp     byte[] get_allSettings( )
java    byte[] get_allSettings( )
uwp     async Task<byte[]> get_allSettings( )
py      get_allSettings( )
php     function get_allSettings( )
es      async get_allSettings( )
cmd     YModule target get_allSettings
```

Useful to backup all the logical names, calibrations parameters, and uploaded files of a device.

**Returns :**
a binary buffer with all the settings.

On failure, throws an exception or returns an binary object of size 0.

## module→get_beacon()
## module→beacon()

YModule

Returns the state of the localization beacon.

| | |
|---|---|
| js | function **get_beacon**( ) |
| cpp | Y_BEACON_enum **get_beacon**( ) |
| m | -(Y_BEACON_enum) beacon |
| pas | Integer **get_beacon**( ): Integer |
| vb | function **get_beacon**( ) As Integer |
| cs | int **get_beacon**( ) |
| dnp | int **get_beacon**( ) |
| java | int **get_beacon**( ) |
| uwp | async Task<int> **get_beacon**( ) |
| py | **get_beacon**( ) |
| php | function **get_beacon**( ) |
| es | async **get_beacon**( ) |
| cmd | 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_errorMessage() YModule
## module→errorMessage()

Returns the error message of the latest error with this module object.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | string **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| es | **get_errorMessage**( ) |

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

**Returns :**
a string corresponding to the latest error message that occurred while using this module object

## module→get_errorType()
## module→errorType()

YModule

Returns the numerical error code of the latest error with this module object.

| | |
|---|---|
| js | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| m | -(YRETCODE) errorType |
| pas | YRETCODE **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | **get_errorType**( ) |
| php | function **get_errorType**( ) |
| es | **get_errorType**( ) |

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

**Returns :**
a number corresponding to the code of the latest error that occurred while using this module object

## module→get_firmwareRelease() YModule
## module→firmwareRelease()

Returns the version of the firmware embedded in the module.

| | |
|---|---|
| js | function **get_firmwareRelease**( ) |
| cpp | string **get_firmwareRelease**( ) |
| m | -(NSString*) firmwareRelease |
| pas | string **get_firmwareRelease**( ): string |
| vb | function **get_firmwareRelease**( ) As String |
| cs | string **get_firmwareRelease**( ) |
| dnp | string **get_firmwareRelease**( ) |
| java | String **get_firmwareRelease**( ) |
| uwp | async Task<string> **get_firmwareRelease**( ) |
| py | **get_firmwareRelease**( ) |
| php | function **get_firmwareRelease**( ) |
| es | async **get_firmwareRelease**( ) |
| cmd | 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_functionIds()
## module→functionIds()

YModule

Retrieve all hardware identifier that match the type passed in argument.

```
js      function get_functionIds( funType)
cpp     vector<string> get_functionIds( string funType)
m       -(NSMutableArray*) functionIds : (NSString*) funType
pas     TStringArray get_functionIds( funType: string): TStringArray
vb      function get_functionIds( ) As List
cs      List<string> get_functionIds( string funType)
dnp     string[] get_functionIds( string funType)
java    ArrayList<String> get_functionIds( String funType)
uwp     async Task<List<string>> get_functionIds( string funType)
py      get_functionIds( funType)
php     function get_functionIds( $funType)
es      async get_functionIds( funType)
cmd     YModule target get_functionIds funType
```

**Parameters :**

**funType** The type of function (Relay, LightSensor, Voltage,...)

**Returns :**

an array of strings.

## module→get_hardwareId() YModule
## module→hardwareId()

Returns the unique hardware identifier of the module.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| dnp | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| es | async **get_hardwareId**( ) |
| pas | string **get_hardwareId**( ): string |
| uwp | async Task<string> **get_hardwareId**( ) |
| cmd | YModule **target get_hardwareId** |

The unique hardware identifier is made of the device serial number followed by string ".module".

**Returns :**
a string that uniquely identifies the module

## module→get_icon2d()
## module→icon2d()

YModule

Returns the icon of the module.

```
js      function get_icon2d( )
cpp     string get_icon2d( )
m       -(NSMutableData*) icon2d
pas     TByteArray get_icon2d( ): TByteArray
vb      function get_icon2d( ) As Byte
cs      byte[] get_icon2d( )
dnp     byte[] get_icon2d( )
java    byte[] get_icon2d( )
uwp     async Task<byte[]> get_icon2d( )
py      get_icon2d( )
php     function get_icon2d( )
es      async get_icon2d( )
cmd     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()

Returns a string with last logs of the module.

| | |
|---|---|
| js | function **get_lastLogs**( ) |
| cpp | string **get_lastLogs**( ) |
| m | -(NSString*) lastLogs |
| pas | string **get_lastLogs**( ): string |
| vb | function **get_lastLogs**( ) As String |
| cs | string **get_lastLogs**( ) |
| dnp | string **get_lastLogs**( ) |
| java | String **get_lastLogs**( ) |
| uwp | async Task<string> **get_lastLogs**( ) |
| py | **get_lastLogs**( ) |
| php | function **get_lastLogs**( ) |
| es | async **get_lastLogs**( ) |
| cmd | 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()
## module→logicalName()

YModule

Returns the logical name of the module.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | string **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| dnp | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| uwp | async Task<string> **get_logicalName**( ) |
| py | **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| es | async **get_logicalName**( ) |
| cmd | 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()
## module→luminosity()

YModule

Returns the luminosity of the module informative LEDs (from 0 to 100).

```
js      function get_luminosity( )
cpp     int get_luminosity( )
m       -(int) luminosity
pas     LongInt get_luminosity( ): LongInt
vb      function get_luminosity( ) As Integer
cs      int get_luminosity( )
dnp     int get_luminosity( )
java    int get_luminosity( )
uwp     async Task<int> get_luminosity( )
py      get_luminosity( )
php     function get_luminosity( )
es      async get_luminosity( )
cmd     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_parentHub()
## module→parentHub()

YModule

Returns the serial number of the YoctoHub on which this module is connected.

```
js      function get_parentHub( )
cpp     string get_parentHub( )
m       -(NSString*) parentHub
pas     string get_parentHub( ): string
vb      function get_parentHub( ) As String
cs      string get_parentHub( )
dnp     string get_parentHub( )
java    String get_parentHub( )
uwp     async Task<string> get_parentHub( )
py      get_parentHub( )
php     function get_parentHub( )
es      async get_parentHub( )
cmd     YModule target get_parentHub
```

If the module is connected by USB, or if the module is the root YoctoHub, an empty string is returned.

**Returns :**
a string with the serial number of the YoctoHub or an empty string

## module→get_persistentSettings() YModule
## module→persistentSettings()

Returns the current state of persistent module settings.

| | |
|---|---|
| js | function **get_persistentSettings**( ) |
| cpp | Y_PERSISTENTSETTINGS_enum **get_persistentSettings**( ) |
| m | -(Y_PERSISTENTSETTINGS_enum) persistentSettings |
| pas | Integer **get_persistentSettings**( ): Integer |
| vb | function **get_persistentSettings**( ) As Integer |
| cs | int **get_persistentSettings**( ) |
| dnp | int **get_persistentSettings**( ) |
| java | int **get_persistentSettings**( ) |
| uwp | async Task<int> **get_persistentSettings**( ) |
| py | **get_persistentSettings**( ) |
| php | function **get_persistentSettings**( ) |
| es | async **get_persistentSettings**( ) |
| cmd | 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()
## module→productId()

YModule

Returns the USB device identifier of the module.

```
js      function get_productId( )
cpp     int get_productId( )
m       -(int) productId
pas     LongInt get_productId( ): LongInt
vb      function get_productId( ) As Integer
cs      int get_productId( )
dnp     int get_productId( )
java    int get_productId( )
uwp     async Task<int> get_productId( )
py      get_productId( )
php     function get_productId( )
es      async get_productId( )
cmd     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()

Returns the commercial name of the module, as set by the factory.

| | |
|---|---|
| js | function **get_productName**( ) |
| cpp | string **get_productName**( ) |
| m | -(NSString*) productName |
| pas | string **get_productName**( ): string |
| vb | function **get_productName**( ) As String |
| cs | string **get_productName**( ) |
| dnp | string **get_productName**( ) |
| java | String **get_productName**( ) |
| uwp | async Task<string> **get_productName**( ) |
| py | **get_productName**( ) |
| php | function **get_productName**( ) |
| es | async **get_productName**( ) |
| cmd | 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()

Returns the release number of the module hardware, preprogrammed at the factory.

```
js      function get_productRelease( )
cpp     int get_productRelease( )
m       -(int) productRelease
pas     LongInt get_productRelease( ): LongInt
vb      function get_productRelease( ) As Integer
cs      int get_productRelease( )
dnp     int get_productRelease( )
java    int get_productRelease( )
uwp     async Task<int> get_productRelease( )
py      get_productRelease( )
php     function get_productRelease( )
es      async get_productRelease( )
cmd     YModule target get_productRelease
```

The original hardware release returns value 1, revision B returns value 2, etc.

**Returns :**
an integer corresponding to the release number of the module hardware, preprogrammed at the factory

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

## module→get_rebootCountdown() YModule
## module→rebootCountdown()

Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

| | |
|---|---|
| js | function **get_rebootCountdown**( ) |
| cpp | int **get_rebootCountdown**( ) |
| m | -(int) rebootCountdown |
| pas | LongInt **get_rebootCountdown**( ): LongInt |
| vb | function **get_rebootCountdown**( ) As Integer |
| cs | int **get_rebootCountdown**( ) |
| dnp | int **get_rebootCountdown**( ) |
| java | int **get_rebootCountdown**( ) |
| uwp | async Task<int> **get_rebootCountdown**( ) |
| py | **get_rebootCountdown**( ) |
| php | function **get_rebootCountdown**( ) |
| es | async **get_rebootCountdown**( ) |
| cmd | 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()

Returns the serial number of the module, as set by the factory.

```
js      function get_serialNumber( )
cpp     string get_serialNumber( )
m       -(NSString*) serialNumber
pas     string get_serialNumber( ): string
vb      function get_serialNumber( ) As String
cs      string get_serialNumber( )
dnp     string get_serialNumber( )
java    String get_serialNumber( )
uwp     async Task<string> get_serialNumber( )
py      get_serialNumber( )
php     function get_serialNumber( )
es      async get_serialNumber( )
cmd     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_subDevices()
## module→subDevices()

YModule

Returns a list of all the modules that are plugged into the current module.

```
js      function get_subDevices( )
cpp     vector<string> get_subDevices( )
m       -(NSMutableArray*) subDevices
pas     TStringArray get_subDevices( ): TStringArray
vb      function get_subDevices( ) As List
cs      List<string> get_subDevices( )
dnp     string[] get_subDevices( )
java    ArrayList<String> get_subDevices( )
uwp     async Task<List<string>> get_subDevices( )
py      get_subDevices( )
php     function get_subDevices( )
es      async get_subDevices( )
cmd     YModule target get_subDevices
```

This method only makes sense when called for a YoctoHub/VirtualHub. Otherwise, an empty array will be returned.

**Returns :**
an array of strings containing the sub modules.

## module→get_upTime()
## module→upTime()

YModule

Returns the number of milliseconds spent since the module was powered on.

```
js      function get_upTime( )
cpp     s64 get_upTime( )
m       -(s64) upTime
pas     int64 get_upTime( ): int64
vb      function get_upTime( ) As Long
cs      long get_upTime( )
dnp     long get_upTime( )
java    long get_upTime( )
uwp     async Task<long> get_upTime( )
py      get_upTime( )
php     function get_upTime( )
es      async get_upTime( )
cmd     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_url()
## module→url()

YModule

Returns the URL used to access the module.

```
js      function get_url( )
cpp     string get_url( )
m       -(NSString*) url
pas     string get_url( ): string
vb      function get_url( ) As String
cs      string get_url( )
dnp     string get_url( )
java    String get_url( )
uwp     async Task<string> get_url( )
py      get_url( )
php     function get_url( )
es      async get_url( )
cmd     YModule target get_url
```

If the module is connected by USB, the string 'usb' is returned.

**Returns :**
a string with the URL of the module.

## module→get_usbCurrent()
## module→usbCurrent()

YModule

Returns the current consumed by the module on the USB bus, in milli-amps.

```
js      function get_usbCurrent( )
cpp     int get_usbCurrent( )
m       -(int) usbCurrent
pas     LongInt get_usbCurrent( ): LongInt
vb      function get_usbCurrent( ) As Integer
cs      int get_usbCurrent( )
dnp     int get_usbCurrent( )
java    int get_usbCurrent( )
uwp     async Task<int> get_usbCurrent( )
py      get_usbCurrent( )
php     function get_usbCurrent( )
es      async get_usbCurrent( )
cmd     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_userData()
## module→userData()

YModule

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

| | |
|---|---|
| js | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(id) userData |
| pas | Tobject **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | **get_userData**( ) |
| php | function **get_userData**( ) |
| es | async **get_userData**( ) |

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

**Returns :**
the object stored previously by the caller.

## module→get_userVar() YModule
## module→userVar()

Returns the value previously stored in this attribute.

```
js      function get_userVar( )
cpp     int get_userVar( )
m       -(int) userVar
pas     LongInt get_userVar( ): LongInt
vb      function get_userVar( ) As Integer
cs      int get_userVar( )
dnp     int get_userVar( )
java    int get_userVar( )
uwp     async Task<int> get_userVar( )
py      get_userVar( )
php     function get_userVar( )
es      async get_userVar( )
cmd     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→hasFunction() YModule

Tests if the device includes a specific function.

```
js      function hasFunction( funcId)
cpp     bool hasFunction( string funcId)
m       -(bool) hasFunction : (NSString*) funcId
pas     boolean hasFunction( funcId: string): boolean
vb      function hasFunction( ) As Boolean
cs      bool hasFunction( string funcId)
dnp     bool hasFunction( string funcId)
java    boolean hasFunction( String funcId)
uwp     async Task<bool> hasFunction( string funcId)
py      hasFunction( funcId)
php     function hasFunction( $funcId)
es      async hasFunction( funcId)
cmd     YModule target hasFunction funcId
```

This method takes a function identifier and returns a boolean.

**Parameters :**

**funcId** the requested function identifier

**Returns :**

true if the device has the function identifier

## module→isOnline() YModule

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

| | |
|---|---|
| js | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | boolean **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| dnp | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | **isOnline**( ) |
| php | function **isOnline**( ) |
| es | async **isOnline**( ) |

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

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

## module→isOnline_async() YModule

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

```
js  function isOnline_async( callback, context)
```

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

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

**Parameters :**

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

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

**Returns :**

nothing : the result is provided to the callback.

## module→load() YModule

Preloads the module cache with a specified validity duration.

```
js      function load( msValidity)
cpp     YRETCODE load( int msValidity)
m       -(YRETCODE) load : (u64) msValidity
pas     YRETCODE load( msValidity: u64): YRETCODE
vb      function load( ByVal msValidity As Long) As YRETCODE
cs      YRETCODE load( ulong msValidity)
java    int load( long msValidity)
py      load( msValidity)
php     function load( $msValidity)
es      async load( msValidity)
```

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

**Parameters :**

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

**Returns :**

YAPI_SUCCESS when the call succeeds.

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

## module→load_async() YModule

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

```
js  function load_async( msValidity, callback, context)
```

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

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

**Parameters :**

| | |
|---|---|
| **msValidity** | an integer corresponding to the validity of the loaded module parameters, in milliseconds |
| **callback** | callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving module object and the error code (or YAPI_SUCCESS) |
| **context** | caller-specific object that is passed as-is to the callback function |

**Returns :**

nothing : the result is provided to the callback.

## module→log() YModule

Adds a text message to the device logs.

```
js      function log( text)
cpp     int log( string text)
m       -(int) log : (NSString*) text
pas     LongInt log( text: string): LongInt
vb      function log( ) As Integer
cs      int log( string text)
dnp     int log( string text)
java    int log( String text)
uwp     async Task<int> log( string text)
py      log( text)
php     function log( $text)
es      async log( text)
cmd     YModule target log text
```

This function is useful in particular to trace the execution of HTTP callbacks. If a newline is desired after the message, it must be included in the string.

**Parameters :**

**text** the string to append to the logs.

**Returns :**

YAPI_SUCCESS if the call succeeds.

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

## module→nextModule() YModule

Continues the module enumeration started using `yFirstModule()`.

| | |
|---|---|
| js | function **nextModule**( ) |
| cpp | YModule * **nextModule**( ) |
| m | -(YModule*) **nextModule** |
| pas | TYModule **nextModule**( ): TYModule |
| vb | function **nextModule**( ) As YModule |
| cs | YModule **nextModule**( ) |
| java | YModule **nextModule**( ) |
| uwp | YModule **nextModule**( ) |
| py | **nextModule**( ) |
| php | function **nextModule**( ) |
| es | **nextModule**( ) |

Caution: You can't make any assumption about the returned modules order. If you want to find a specific module, use `Module.findModule()` and a hardwareID or a logical name.

**Returns :**

a pointer to a `YModule` object, corresponding to the next module found, or a `null` pointer if there are no more modules to enumerate.

## module→reboot() YModule

Schedules a simple module reboot after the given number of seconds.

```
js      function reboot( secBeforeReboot)
cpp     int reboot( int secBeforeReboot)
m       -(int) reboot : (int) secBeforeReboot
pas     LongInt reboot( secBeforeReboot: LongInt): LongInt
vb      function reboot( ) As Integer
cs      int reboot( int secBeforeReboot)
dnp     int reboot( int secBeforeReboot)
java    int reboot( int secBeforeReboot)
uwp     async Task<int> reboot( int secBeforeReboot)
py      reboot( secBeforeReboot)
php     function reboot( $secBeforeReboot)
es      async reboot( secBeforeReboot)
cmd     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→registerBeaconCallback() YModule

Register a callback function, to be called when the localization beacon of the module has been changed.

| | |
|---|---|
| js | function **registerBeaconCallback**( **callback**) |
| cpp | int **registerBeaconCallback**( YModuleBeaconCallback **callback**) |
| m | -(int) **registerBeaconCallback** : (YModuleBeaconCallback) **callback** |
| pas | LongInt **registerBeaconCallback**( **callback**: TYModuleBeaconCallback): LongInt |
| vb | function **registerBeaconCallback**( ) As Integer |
| cs | int **registerBeaconCallback**( BeaconCallback **callback**) |
| java | int **registerBeaconCallback**( BeaconCallback **callback**) |
| uwp | async Task<int> **registerBeaconCallback**( BeaconCallback **callback**) |
| py | **registerBeaconCallback**( **callback**) |
| php | function **registerBeaconCallback**( **$callback**) |
| es | async **registerBeaconCallback**( **callback**) |

The callback function should take two arguments: the YModule object of which the beacon has changed, and an integer describing the new beacon state.

**Parameters :**

**callback** The callback function to call, or `null` to unregister a

## module→registerConfigChangeCallback() YModule

Register a callback function, to be called when a persistent settings in a device configuration has been changed (e.g.

```
js    function registerConfigChangeCallback( callback)
cpp   int registerConfigChangeCallback( YModuleConfigChangeCallback callback)
m     -(int) registerConfigChangeCallback : (YModuleConfigChangeCallback) callback
pas   LongInt registerConfigChangeCallback( callback: TYModuleConfigChangeCallback): LongInt
vb    function registerConfigChangeCallback( ) As Integer
cs    int registerConfigChangeCallback( ConfigChangeCallback callback)
java  int registerConfigChangeCallback( ConfigChangeCallback callback)
uwp   async Task<int> registerConfigChangeCallback( ConfigChangeCallback callback)
py    registerConfigChangeCallback( callback)
php   function registerConfigChangeCallback( $callback)
es    async registerConfigChangeCallback( callback)
```

change of unit, etc).

**Parameters :**

**callback** a procedure taking a YModule parameter, or `null`

## module→registerLogCallback() YModule

Registers a device log callback function.

| | |
|---|---|
| js | function **registerLogCallback**( **callback**) |
| cpp | int **registerLogCallback**( YModuleLogCallback **callback**) |
| m | -(int) **registerLogCallback** : (YModuleLogCallback) **callback** |
| pas | LongInt **registerLogCallback**( **callback**: TYModuleLogCallback): LongInt |
| vb | function **registerLogCallback**( ) As Integer |
| cs | int **registerLogCallback**( LogCallback **callback**) |
| java | int **registerLogCallback**( LogCallback **callback**) |
| uwp | async Task<int> **registerLogCallback**( LogCallback **callback**) |
| py | **registerLogCallback**( **callback**) |
| php | function **registerLogCallback**( **$callback**) |
| es | async **registerLogCallback**( **callback**) |

This callback will be called each time that a module sends a new log message. Mostly useful to debug a Yoctopuce module.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the module object that emitted the log message, and the character string containing the log.

## module→revertFromFlash() YModule

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

```
js      function revertFromFlash( )
cpp     int revertFromFlash( )
m       -(int) revertFromFlash
pas     LongInt revertFromFlash( ): LongInt
vb      function revertFromFlash( ) As Integer
cs      int revertFromFlash( )
dnp     int revertFromFlash( )
java    int revertFromFlash( )
uwp     async Task<int> revertFromFlash( )
py      revertFromFlash( )
php     function revertFromFlash( )
es      async revertFromFlash( )
cmd     YModule target revertFromFlash
```

**Returns :**
YAPI_SUCCESS when the call succeeds.

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

## module→saveToFlash() YModule

Saves current settings in the nonvolatile memory of the module.

| | |
|---|---|
| js | function **saveToFlash**( ) |
| cpp | int **saveToFlash**( ) |
| m | -(int) **saveToFlash** |
| pas | LongInt **saveToFlash**( ): LongInt |
| vb | function **saveToFlash**( ) As Integer |
| cs | int **saveToFlash**( ) |
| dnp | int **saveToFlash**( ) |
| java | int **saveToFlash**( ) |
| uwp | async Task<int> **saveToFlash**( ) |
| py | **saveToFlash**( ) |
| php | function **saveToFlash**( ) |
| es | async **saveToFlash**( ) |
| cmd | 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()

Restores all the settings of the device.

```
js    function set_allSettings( settings)
cpp   int set_allSettings( string settings)
m     -(int) setAllSettings : (NSData*) settings
pas   LongInt set_allSettings( settings: TByteArray): LongInt
vb    procedure set_allSettings( )
cs    int set_allSettings( )
dnp   int set_allSettings( )
java  int set_allSettings( byte[] settings)
uwp   async Task<int> set_allSettings( )
py    set_allSettings( settings)
php   function set_allSettings( $settings)
es    async set_allSettings( settings)
cmd   YModule target set_allSettings settings
```

Useful to restore all the logical names and calibrations parameters of a module from a backup.Remember to call the `saveToFlash()` method of the module if the modifications must be kept.

**Parameters :**

**settings** a binary buffer with all the settings.

**Returns :**

`YAPI_SUCCESS` when the call succeeds.

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

## module→set_allSettingsAndFiles() YModule
## module→setAllSettingsAndFiles()

Restores all the settings and uploaded files to the module.

| | |
|---|---|
| js | function **set_allSettingsAndFiles**( **settings**) |
| cpp | int **set_allSettingsAndFiles**( string **settings**) |
| m | -(int) setAllSettingsAndFiles : (NSData*) **settings** |
| pas | LongInt **set_allSettingsAndFiles**( **settings**: TByteArray): LongInt |
| vb | procedure **set_allSettingsAndFiles**( ) |
| cs | int **set_allSettingsAndFiles**( ) |
| dnp | int **set_allSettingsAndFiles**( ) |
| java | int **set_allSettingsAndFiles**( byte[] **settings**) |
| uwp | async Task<int> **set_allSettingsAndFiles**( ) |
| py | **set_allSettingsAndFiles**( **settings**) |
| php | function **set_allSettingsAndFiles**( **$settings**) |
| es | async **set_allSettingsAndFiles**( **settings**) |
| cmd | YModule **target set_allSettingsAndFiles settings** |

This method is useful to restore all the logical names and calibrations parameters, uploaded files etc. of a device from a backup. Remember to call the `saveToFlash()` method of the module if the modifications must be kept.

**Parameters :**

**settings** a binary buffer with all the settings.

**Returns :**

`YAPI_SUCCESS` when the call succeeds.

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

## module→set_beacon()
## module→setBeacon()

YModule

Turns on or off the module localization beacon.

```
js      function set_beacon( newval)
cpp     int set_beacon( Y_BEACON_enum newval)
m       -(int) setBeacon : (Y_BEACON_enum) newval
pas     integer set_beacon( newval: Integer): integer
vb      function set_beacon( ByVal newval As Integer) As Integer
cs      int set_beacon( int newval)
dnp     int set_beacon( int newval)
java    int set_beacon( int newval)
uwp     async Task<int> set_beacon( int newval)
py      set_beacon( newval)
php     function set_beacon( $newval)
es      async set_beacon( newval)
cmd     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()

Changes the logical name of the module.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| cpp | int **set_logicalName**( string **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | integer **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| dnp | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| uwp | async Task<int> **set_logicalName**( string **newval**) |
| py | **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| es | async **set_logicalName**( **newval**) |
| cmd | 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()
## module→setLuminosity()

YModule

Changes the luminosity of the module informative leds.

```
js      function set_luminosity( newval)
cpp     int set_luminosity( int newval)
m       -(int) setLuminosity : (int) newval
pas     integer set_luminosity( newval: LongInt): integer
vb      function set_luminosity( ByVal newval As Integer) As Integer
cs      int set_luminosity( int newval)
dnp     int set_luminosity( int newval)
java    int set_luminosity( int newval)
uwp     async Task<int> set_luminosity( int newval)
py      set_luminosity( newval)
php     function set_luminosity( $newval)
es      async set_luminosity( newval)
cmd     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_userData() YModule
## module→setUserData()

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

| | |
|---|---|
| js | function **set_userData**( **data**) |
| cpp | void **set_userData**( void * **data**) |
| m | -(void) setUserData : (id) **data** |
| pas | **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| es | async **set_userData**( **data**) |

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

**Parameters :**

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

## module→set_userVar()
## module→setUserVar()

YModule

Stores a 32 bit value in the device RAM.

```
js      function set_userVar( newval)
cpp     int set_userVar( int newval)
m       -(int) setUserVar : (int) newval
pas     integer set_userVar( newval: LongInt): integer
vb      function set_userVar( ByVal newval As Integer) As Integer
cs      int set_userVar( int newval)
dnp     int set_userVar( int newval)
java    int set_userVar( int newval)
uwp     async Task<int> set_userVar( int newval)
py      set_userVar( newval)
php     function set_userVar( $newval)
es      async set_userVar( newval)
cmd     YModule target set_userVar newval
```

This attribute is at programmer disposal, should he need to store a state variable. 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→triggerConfigChangeCallback() YModule

Triggers a configuration change callback, to check if they are supported or not.

| | |
|---|---|
| js | function **triggerConfigChangeCallback**( ) |
| cpp | int **triggerConfigChangeCallback**( ) |
| m | -(int) **triggerConfigChangeCallback** |
| pas | LongInt **triggerConfigChangeCallback**( ): LongInt |
| vb | function **triggerConfigChangeCallback**( ) As Integer |
| cs | int **triggerConfigChangeCallback**( ) |
| dnp | int **triggerConfigChangeCallback**( ) |
| java | int **triggerConfigChangeCallback**( ) |
| uwp | async Task<int> **triggerConfigChangeCallback**( ) |
| py | **triggerConfigChangeCallback**( ) |
| php | function **triggerConfigChangeCallback**( ) |
| es | async **triggerConfigChangeCallback**( ) |
| cmd | YModule **target triggerConfigChangeCallback** |

## module→triggerFirmwareUpdate() YModule

Schedules a module reboot into special firmware update mode.

```
js    function triggerFirmwareUpdate( secBeforeReboot)
cpp   int triggerFirmwareUpdate( int secBeforeReboot)
m     -(int) triggerFirmwareUpdate : (int) secBeforeReboot
pas   LongInt triggerFirmwareUpdate( secBeforeReboot: LongInt): LongInt
vb    function triggerFirmwareUpdate( ) As Integer
cs    int triggerFirmwareUpdate( int secBeforeReboot)
dnp   int triggerFirmwareUpdate( int secBeforeReboot)
java  int triggerFirmwareUpdate( int secBeforeReboot)
uwp   async Task<int> triggerFirmwareUpdate( int secBeforeReboot)
py    triggerFirmwareUpdate( secBeforeReboot)
php   function triggerFirmwareUpdate( $secBeforeReboot)
es    async triggerFirmwareUpdate( secBeforeReboot)
cmd   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

Prepares a firmware update of the module.

| | |
|---|---|
| js | function **updateFirmware**( **path**) |
| cpp | YFirmwareUpdate **updateFirmware**( string **path**) |
| m | -(YFirmwareUpdate*) **updateFirmware** : (NSString*) **path** |
| pas | TYFirmwareUpdate **updateFirmware**( **path**: string): TYFirmwareUpdate |
| vb | function **updateFirmware**( ) As YFirmwareUpdate |
| cs | YFirmwareUpdate **updateFirmware**( string **path**) |
| dnp | YFirmwareUpdateProxy **updateFirmware**( string **path**) |
| java | YFirmwareUpdate **updateFirmware**( String **path**) |
| uwp | async Task<YFirmwareUpdate> **updateFirmware**( string **path**) |
| py | **updateFirmware**( **path**) |
| php | function **updateFirmware**( **$path**) |
| es | async **updateFirmware**( **path**) |
| cmd | YModule **target updateFirmware path** |

This method returns a `YFirmwareUpdate` object which handles the firmware update process.

**Parameters :**

**path** the path of the `.byn` file to use.

**Returns :**

a `YFirmwareUpdate` object or NULL on error.

## module→updateFirmwareEx() YModule

Prepares a firmware update of the module.

```
js      function updateFirmwareEx( path, force)
cpp     YFirmwareUpdate updateFirmwareEx( string path, bool force)
m       -(YFirmwareUpdate*) updateFirmwareEx : (NSString*) path
                                             : (bool) force
pas     TYFirmwareUpdate updateFirmwareEx( path: string, force: boolean): TYFirmwareUpdate
vb      function updateFirmwareEx( ) As YFirmwareUpdate
cs      YFirmwareUpdate updateFirmwareEx( string path, bool force)
dnp     YFirmwareUpdateProxy updateFirmwareEx( string path, bool force)
java    YFirmwareUpdate updateFirmwareEx( String path, boolean force)
uwp     async Task<YFirmwareUpdate> updateFirmwareEx( string path, bool force)
py      updateFirmwareEx( path, force)
php     function updateFirmwareEx( $path, $force)
es      async updateFirmwareEx( path, force)
cmd     YModule target updateFirmwareEx path force
```

This method returns a `YFirmwareUpdate` object which handles the firmware update process.

**Parameters :**

**path** the path of the `.byn` file to use.

**force** true to force the firmware update even if some prerequisites appear not to be met

**Returns :**

a `YFirmwareUpdate` object or NULL on error.

## module→wait_async() YModule

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

```
js  function wait_async( callback, context)
es  wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

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

**Returns :**

nothing.

# 22.3. Class YGps

Geolocalization control interface (GPS, GNSS, ...), available for instance in the Yocto-GPS

The YGps class allows you to retrieve positioning data from a GPS/GNSS sensor. This class can provides complete positioning information. However, if you wish to define callbacks on position changes or record the position in the datalogger, you should use the YLatitude et YLongitude classes.

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

| | |
|---|---|
| es | in HTML: <script src="../../lib/yocto_gps.js"></script><br>in node.js: require('yoctolib-es2017/yocto_gps.js'); |
| js | <script type='text/javascript' src='yocto_gps.js'></script> |
| cpp | #include "yocto_gps.h" |
| m | #import "yocto_gps.h" |
| pas | uses yocto_gps; |
| vb | yocto_gps.vb |
| cs | yocto_gps.cs |
| dnp | import YoctoProxyAPI.YGpsProxy |
| java | import com.yoctopuce.YoctoAPI.YGps; |
| uwp | import com.yoctopuce.YoctoAPI.YGps; |
| py | from yocto_gps import * |
| php | require_once('yocto_gps.php'); |
| vi | YGps.vi |

## Global functions

**YGps.FindGps**(**func**)

Retrieves a geolocalization module for a given identifier.

**YGps.FindGpsInContext**(**yctx**, **func**)

Retrieves a geolocalization module for a given identifier in a YAPI context.

**YGps.FirstGps**()

Starts the enumeration of geolocalization modules currently accessible.

**YGps.FirstGpsInContext**(**yctx**)

Starts the enumeration of geolocalization modules currently accessible.

**YGps.GetSimilarFunctions**()

Enumerates all functions of type Gps available on the devices currently reachable by the library, and returns their unique hardware ID.

## YGps properties

**gps→AdvertisedValue** *[read-only]*

Short string representing the current state of the function.

**gps→CoordSystem** *[writable]*

Representation system used for positioning data.

**gps→FriendlyName** *[read-only]*

Global identifier of the function in the format MODULE_NAME.FUNCTION_NAME.

**gps→FunctionId** *[read-only]*

Hardware identifier of the geolocalization module, without reference to the module.

**gps→HardwareId** *[read-only]*

Unique hardware identifier of the function in the form SERIAL.FUNCTIONID.

**gps→IsFixed** *[read-only]*

TRUE if the receiver has found enough satellites to work.

**gps→IsOnline** *[read-only]*

Checks if the function is currently reachable.

**gps→LogicalName** *[writable]*

Logical name of the function.

**gps→SatCount** *[read-only]*

Total count of satellites used to compute GPS position.

**gps→SerialNumber** *[read-only]*

Serial number of the module, as set by the factory.

**gps→UtcOffset** *[writable]*

Number of seconds between current time and UTC time (time zone).

### YGps methods

**gps→clearCache**()

Invalidates the cache.

**gps→describe**()

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

**gps→get_advertisedValue**()

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

**gps→get_altitude**()

Returns the current altitude.

**gps→get_constellation**()

Returns the the satellites constellation used to compute positioning data.

**gps→get_coordSystem**()

Returns the representation system used for positioning data.

**gps→get_dateTime**()

Returns the current time in the form "YYYY/MM/DD hh:mm:ss".

**gps→get_dilution**()

Returns the current horizontal dilution of precision, the smaller that number is, the better .

**gps→get_direction**()

Returns the current move bearing in degrees, zero is the true (geographic) north.

**gps→get_errorMessage**()

Returns the error message of the latest error with the geolocalization module.

**gps→get_errorType**()

Returns the numerical error code of the latest error with the geolocalization module.

**gps→get_friendlyName**()

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

**gps→get_functionDescriptor**()

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

**gps→get_functionId**()

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

**gps→get_gpsRefreshRate**()

Returns effective GPS data refresh frequency.

**gps→get_groundSpeed**()

Returns the current ground speed in Km/h.

**gps→get_hardwareId**()

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

**gps→get_isFixed**()

Returns TRUE if the receiver has found enough satellites to work.

**gps→get_latitude**()

Returns the current latitude.

**gps→get_logicalName**()

Returns the logical name of the geolocalization module.

**gps→get_longitude**()

Returns the current longitude.

**gps→get_module**()

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

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

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

**gps→get_satCount**()

Returns the total count of satellites used to compute GPS position.

**gps→get_satPerConst**()

Returns the count of visible satellites per constellation encoded on a 32 bit integer: bits 0..

**gps→get_serialNumber**()

Returns the serial number of the module, as set by the factory.

**gps→get_unixTime**()

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

**gps→get_userData**()

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

**gps→get_utcOffset**()

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

**gps→isOnline**()

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

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

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

**gps→isReadOnly**()

Test if the function is readOnly.

**gps→load**(**msValidity**)

Preloads the geolocalization module cache with a specified validity duration.

**gps→loadAttribute**(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

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

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

**gps→muteValueCallbacks**()

Disables the propagation of every new advertised value to the parent hub.

**gps→nextGps**()

Continues the enumeration of geolocalization modules started using `yFirstGps()`.

**gps→registerValueCallback**(**callback**)

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

**gps→set_constellation**(**newval**)

Changes the satellites constellation used to compute positioning data.

**gps→set_coordSystem**(**newval**)

Changes the representation system used for positioning data.

**gps→set_logicalName**(**newval**)

Changes the logical name of the geolocalization module.

**gps→set_userData**(**data**)

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

**gps→set_utcOffset**(**newval**)

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

**gps→unmuteValueCallbacks**()

Re-enables the propagation of every new advertised value to the parent hub.

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

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

## YGps.FindGps() YGps
## YGps.FindGps()

Retrieves a geolocalization module for a given identifier.

```
js      function yFindGps( func)
cpp     YGps* yFindGps( string func)
m       +(YGps*) FindGps : (NSString*) func
pas     TYGps yFindGps( func: string): TYGps
vb      function yFindGps( ByVal func As String) As YGps
cs      static YGps FindGps( string func)
dnp     static YGpsProxy FindGps( string func)
java    static YGps FindGps( String func)
uwp     static YGps FindGps( string func)
py      FindGps( func)
php     function yFindGps( $func)
es      static FindGps( func)
```

The identifier can be specified using several formats:

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

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

If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

**Parameters :**

**func** a string that uniquely characterizes the geolocalization module, for instance `YGNSSMK1.gps`.

**Returns :**

a `YGps` object allowing you to drive the geolocalization module.

## YGps.FindGpsInContext() YGps
## YGps.FindGpsInContext()

Retrieves a geolocalization module for a given identifier in a YAPI context.

java static YGps **FindGpsInContext**( YAPIContext **yctx**, String **func**)
uwp static YGps **FindGpsInContext**( YAPIContext **yctx**, string **func**)
es static **FindGpsInContext**( **yctx**, **func**)

The identifier can be specified using several formats:

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

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

**Parameters :**

**yctx** a YAPI context

**func** a string that uniquely characterizes the geolocalization module, for instance `YGNSSMK1.gps`.

**Returns :**

a `YGps` object allowing you to drive the geolocalization module.

## YGps.FirstGps() YGps
## YGps.FirstGps()

Starts the enumeration of geolocalization modules currently accessible.

| | |
|---|---|
| js | function **yFirstGps**( ) |
| cpp | YGps * **yFirstGps**( ) |
| m | +(YGps*) **FirstGps** |
| pas | TYGps **yFirstGps**( ): TYGps |
| vb | function **yFirstGps**( ) As YGps |
| cs | static YGps **FirstGps**( ) |
| java | static YGps **FirstGps**( ) |
| uwp | static YGps **FirstGps**( ) |
| py | **FirstGps**( ) |
| php | function **yFirstGps**( ) |
| es | static **FirstGps**( ) |

Use the method `YGps.nextGps()` to iterate on next geolocalization modules.

**Returns :**

a pointer to a `YGps` object, corresponding to the first geolocalization module currently online, or a `null` pointer if there are none.

## YGps.FirstGpsInContext() YGps
## YGps.FirstGpsInContext()

Starts the enumeration of geolocalization modules currently accessible.

java static YGps **FirstGpsInContext**( YAPIContext **yctx**)
uwp static YGps **FirstGpsInContext**( YAPIContext **yctx**)
es static **FirstGpsInContext**( **yctx**)

Use the method `YGps.nextGps()` to iterate on next geolocalization modules.

**Parameters :**

**yctx** a YAPI context.

**Returns :**

a pointer to a `YGps` object, corresponding to the first geolocalization module currently online, or a `null` pointer if there are none.

## YGps.GetSimilarFunctions() YGps
## YGps.GetSimilarFunctions()

Enumerates all functions of type Gps available on the devices currently reachable by the library, and returns their unique hardware ID.

dnp static new string[] **GetSimilarFunctions**( )

Each of these IDs can be provided as argument to the method `YGps.FindGps` to obtain an object that can control the corresponding device.

**Returns :**
an array of strings, each string containing the unique hardwareId of a device function currently connected.

## gps→AdvertisedValue YGps

Short string representing the current state of the function.

dnp string **AdvertisedValue**

## gps→CoordSystem YGps

Representation system used for positioning data.

dnp int **CoordSystem**

**Possible values:**

Y_COORDSYSTEM_INVALID = 0
Y_COORDSYSTEM_GPS_DMS = 1
Y_COORDSYSTEM_GPS_DM = 2
Y_COORDSYSTEM_GPS_D = 3

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

## gps→FriendlyName YGps

Global identifier of the function in the format MODULE_NAME.FUNCTION_NAME.

dnp string **FriendlyName**

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

## gps→FunctionId YGps

Hardware identifier of the geolocalization module, without reference to the module.

dnp string **FunctionId**

For example `relay1`

## gps→HardwareId — YGps

Unique hardware identifier of the function in the form SERIAL.FUNCTIONID.

dnp string **HardwareId**

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

## gps→IsFixed YGps

TRUE if the receiver has found enough satellites to work.

dnp int **IsFixed**

**Possible values:**

```
Y_ISFIXED_INVALID = 0
Y_ISFIXED_FALSE   = 1
Y_ISFIXED_TRUE    = 2
```

## gps→IsOnline YGps

Checks if the function is currently reachable.

dnp bool **IsOnline**

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

## gps→LogicalName YGps

Logical name of the function.

dnp string **LogicalName**

**Writable**. 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.

## gps→SatCount YGps

Total count of satellites used to compute GPS position.

dnp long **SatCount**

## gps→SerialNumber YGps

Serial number of the module, as set by the factory.

dnp string **SerialNumber**

## gps→UtcOffset YGps

Number of seconds between current time and UTC time (time zone).

dnp int **UtcOffset**

**Writable**. The timezone is automatically rounded to the nearest multiple of 15 minutes. If current UTC time is known, the current time is automatically be updated according to the selected time zone. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

## gps→clearCache() YGps

Invalidates the cache.

| | |
|---|---|
| js | function **clearCache**( ) |
| cpp | void **clearCache**( ) |
| m | -(void) **clearCache** |
| pas | **clearCache**( ) |
| vb | procedure **clearCache**( ) |
| cs | void **clearCache**( ) |
| java | void **clearCache**( ) |
| py | **clearCache**( ) |
| php | function **clearCache**( ) |
| es | async **clearCache**( ) |

Invalidates the cache of the geolocalization module attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

## gps→describe() YGps

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

```
js    function describe( )
cpp   string describe( )
m     -(NSString*) describe
pas   string describe( ): string
vb    function describe( ) As String
cs    string describe( )
java  String describe( )
py    describe( )
php   function describe( )
es    async describe( )
```

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

**Returns :**
a string that describes the geolocalization module (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

## gps→get_advertisedValue() YGps
## gps→advertisedValue()

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

```
js      function get_advertisedValue( )
cpp     string get_advertisedValue( )
m       -(NSString*) advertisedValue
pas     string get_advertisedValue( ): string
vb      function get_advertisedValue( ) As String
cs      string get_advertisedValue( )
dnp     string get_advertisedValue( )
java    String get_advertisedValue( )
uwp     async Task<string> get_advertisedValue( )
py      get_advertisedValue( )
php     function get_advertisedValue( )
es      async get_advertisedValue( )
cmd     YGps target get_advertisedValue
```

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

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

## gps→get_altitude()
## gps→altitude()

YGps

Returns the current altitude.

| | |
|---|---|
| js | function **get_altitude**( ) |
| cpp | double **get_altitude**( ) |
| m | -(double) altitude |
| pas | double **get_altitude**( ): double |
| vb | function **get_altitude**( ) As Double |
| cs | double **get_altitude**( ) |
| dnp | double **get_altitude**( ) |
| java | double **get_altitude**( ) |
| uwp | async Task<double> **get_altitude**( ) |
| py | **get_altitude**( ) |
| php | function **get_altitude**( ) |
| es | async **get_altitude**( ) |
| cmd | YGps **target** **get_altitude** |

Beware: GPS technology is very inaccurate regarding altitude.

**Returns :**
a floating point number corresponding to the current altitude

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

## gps→get_constellation()
## gps→constellation()

YGps

Returns the the satellites constellation used to compute positioning data.

| | |
|---|---|
| js | function **get_constellation**( ) |
| cpp | Y_CONSTELLATION_enum **get_constellation**( ) |
| m | -(Y_CONSTELLATION_enum) constellation |
| pas | Integer **get_constellation**( ): Integer |
| vb | function **get_constellation**( ) As Integer |
| cs | int **get_constellation**( ) |
| dnp | int **get_constellation**( ) |
| java | int **get_constellation**( ) |
| uwp | async Task<int> **get_constellation**( ) |
| py | **get_constellation**( ) |
| php | function **get_constellation**( ) |
| es | async **get_constellation**( ) |
| cmd | YGps **target get_constellation** |

**Returns :**

a value among `Y_CONSTELLATION_GNSS`, `Y_CONSTELLATION_GPS`, `Y_CONSTELLATION_GLONASS`, `Y_CONSTELLATION_GALILEO`, `Y_CONSTELLATION_GPS_GLONASS`, `Y_CONSTELLATION_GPS_GALILEO` and `Y_CONSTELLATION_GLONASS_GALILEO` corresponding to the the satellites constellation used to compute positioning data

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

## gps→get_coordSystem()
## gps→coordSystem()

YGps

Returns the representation system used for positioning data.

| | |
|---|---|
| js | function **get_coordSystem**( ) |
| cpp | Y_COORDSYSTEM_enum **get_coordSystem**( ) |
| m | -(Y_COORDSYSTEM_enum) coordSystem |
| pas | Integer **get_coordSystem**( ): Integer |
| vb | function **get_coordSystem**( ) As Integer |
| cs | int **get_coordSystem**( ) |
| dnp | int **get_coordSystem**( ) |
| java | int **get_coordSystem**( ) |
| uwp | async Task<int> **get_coordSystem**( ) |
| py | **get_coordSystem**( ) |
| php | function **get_coordSystem**( ) |
| es | async **get_coordSystem**( ) |
| cmd | YGps **target get_coordSystem** |

**Returns :**

a value among `Y_COORDSYSTEM_GPS_DMS`, `Y_COORDSYSTEM_GPS_DM` and `Y_COORDSYSTEM_GPS_D` corresponding to the representation system used for positioning data

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

## gps→get_dateTime()
## gps→dateTime()

YGps

Returns the current time in the form "YYYY/MM/DD hh:mm:ss".

```
js      function get_dateTime( )
cpp     string get_dateTime( )
m       -(NSString*) dateTime
pas     string get_dateTime( ): string
vb      function get_dateTime( ) As String
cs      string get_dateTime( )
dnp     string get_dateTime( )
java    String get_dateTime( )
uwp     async Task<string> get_dateTime( )
py      get_dateTime( )
php     function get_dateTime( )
es      async get_dateTime( )
cmd     YGps target get_dateTime
```

**Returns :**
a string corresponding to the current time in the form "YYYY/MM/DD hh:mm:ss"

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

## gps→get_dilution()
## gps→dilution()

YGps

Returns the current horizontal dilution of precision, the smaller that number is, the better .

```
js      function get_dilution( )
cpp     double get_dilution( )
m       -(double) dilution
pas     double get_dilution( ): double
vb      function get_dilution( ) As Double
cs      double get_dilution( )
dnp     double get_dilution( )
java    double get_dilution( )
uwp     async Task<double> get_dilution( )
py      get_dilution( )
php     function get_dilution( )
es      async get_dilution( )
cmd     YGps target get_dilution
```

**Returns :**

a floating point number corresponding to the current horizontal dilution of precision, the smaller that number is, the better

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

## gps→get_direction()
## gps→direction()

YGps

Returns the current move bearing in degrees, zero is the true (geographic) north.

```
js      function get_direction( )
cpp     double get_direction( )
m       -(double) direction
pas     double get_direction( ): double
vb      function get_direction( ) As Double
cs      double get_direction( )
dnp     double get_direction( )
java    double get_direction( )
uwp     async Task<double> get_direction( )
py      get_direction( )
php     function get_direction( )
es      async get_direction( )
cmd     YGps target get_direction
```

**Returns :**

a floating point number corresponding to the current move bearing in degrees, zero is the true (geographic) north

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

## gps→get_errorMessage() YGps
## gps→errorMessage()

Returns the error message of the latest error with the geolocalization module.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | string **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| es | **get_errorMessage**( ) |

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

**Returns :**
a string corresponding to the latest error message that occured while using the geolocalization module object

## gps→get_errorType() YGps
## gps→errorType()

Returns the numerical error code of the latest error with the geolocalization module.

| | |
|---|---|
| js | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| m | -(YRETCODE) errorType |
| pas | YRETCODE **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | **get_errorType**( ) |
| php | function **get_errorType**( ) |
| es | **get_errorType**( ) |

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

**Returns :**
a number corresponding to the code of the latest error that occurred while using the geolocalization module object

## gps→get_friendlyName() YGps
## gps→friendlyName()

Returns a global identifier of the geolocalization module in the format MODULE_NAME.FUNCTION_NAME.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| dnp | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| es | async **get_friendlyName**( ) |

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

**Returns :**

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

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

## gps→get_functionDescriptor() YGps
## gps→functionDescriptor()

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

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | YFUN_DESCR **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| es | async **get_functionDescriptor**( ) |

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

**Returns :**
an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

## gps→get_functionId()
## gps→functionId()

YGps

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

| | |
|---|---|
| js | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| dnp | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | **get_functionId**( ) |
| php | function **get_functionId**( ) |
| es | async **get_functionId**( ) |

For example `relay1`

**Returns :**
a string that identifies the geolocalization module (ex: `relay1`)

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

## gps→get_gpsRefreshRate() YGps
## gps→gpsRefreshRate()

Returns effective GPS data refresh frequency.

```
js      function get_gpsRefreshRate( )
cpp     double get_gpsRefreshRate( )
m       -(double) gpsRefreshRate
pas     double get_gpsRefreshRate( ): double
vb      function get_gpsRefreshRate( ) As Double
cs      double get_gpsRefreshRate( )
dnp     double get_gpsRefreshRate( )
java    double get_gpsRefreshRate( )
uwp     async Task<double> get_gpsRefreshRate( )
py      get_gpsRefreshRate( )
php     function get_gpsRefreshRate( )
es      async get_gpsRefreshRate( )
cmd     YGps target get_gpsRefreshRate
```

this value is refreshed every 5 seconds only.

**Returns :**
a floating point number corresponding to effective GPS data refresh frequency

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

## gps→get_groundSpeed() YGps
## gps→groundSpeed()

Returns the current ground speed in Km/h.

```
js      function get_groundSpeed( )
cpp     double get_groundSpeed( )
m       -(double) groundSpeed
pas     double get_groundSpeed( ): double
vb      function get_groundSpeed( ) As Double
cs      double get_groundSpeed( )
dnp     double get_groundSpeed( )
java    double get_groundSpeed( )
uwp     async Task<double> get_groundSpeed( )
py      get_groundSpeed( )
php     function get_groundSpeed( )
es      async get_groundSpeed( )
cmd     YGps target get_groundSpeed
```

**Returns :**
a floating point number corresponding to the current ground speed in Km/h

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

## gps→get_hardwareId() YGps
## gps→hardwareId()

Returns the unique hardware identifier of the geolocalization module in the form SERIAL.FUNCTIONID.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| dnp | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| es | async **get_hardwareId**( ) |

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

**Returns :**
a string that uniquely identifies the geolocalization module (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

## gps→get_isFixed()
## gps→isFixed()

YGps

Returns TRUE if the receiver has found enough satellites to work.

| | |
|---|---|
| js | function **get_isFixed**( ) |
| cpp | Y_ISFIXED_enum **get_isFixed**( ) |
| m | -(Y_ISFIXED_enum) isFixed |
| pas | Integer **get_isFixed**( ): Integer |
| vb | function **get_isFixed**( ) As Integer |
| cs | int **get_isFixed**( ) |
| dnp | int **get_isFixed**( ) |
| java | int **get_isFixed**( ) |
| uwp | async Task<int> **get_isFixed**( ) |
| py | **get_isFixed**( ) |
| php | function **get_isFixed**( ) |
| es | async **get_isFixed**( ) |
| cmd | YGps **target** **get_isFixed** |

**Returns :**

either `Y_ISFIXED_FALSE` or `Y_ISFIXED_TRUE`, according to TRUE if the receiver has found enough satellites to work

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

## gps→get_latitude()
## gps→latitude()

YGps

Returns the current latitude.

```
js      function get_latitude( )
cpp     string get_latitude( )
m       -(NSString*) latitude
pas     string get_latitude( ): string
vb      function get_latitude( ) As String
cs      string get_latitude( )
dnp     string get_latitude( )
java    String get_latitude( )
uwp     async Task<string> get_latitude( )
py      get_latitude( )
php     function get_latitude( )
es      async get_latitude( )
cmd     YGps target get_latitude
```

**Returns :**
a string corresponding to the current latitude

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

## gps→get_logicalName()
## gps→logicalName()

YGps

Returns the logical name of the geolocalization module.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | string **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| dnp | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| uwp | async Task<string> **get_logicalName**( ) |
| py | **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| es | async **get_logicalName**( ) |
| cmd | YGps **target get_logicalName** |

**Returns :**

a string corresponding to the logical name of the geolocalization module.

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

## gps→get_longitude()
## gps→longitude()

YGps

Returns the current longitude.

```
js      function get_longitude( )
cpp     string get_longitude( )
m       -(NSString*) longitude
pas     string get_longitude( ): string
vb      function get_longitude( ) As String
cs      string get_longitude( )
dnp     string get_longitude( )
java    String get_longitude( )
uwp     async Task<string> get_longitude( )
py      get_longitude( )
php     function get_longitude( )
es      async get_longitude( )
cmd     YGps target get_longitude
```

**Returns :**
a string corresponding to the current longitude

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

## gps→get_module()
## gps→module()

YGps

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

| | |
|---|---|
| js | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | TYModule **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| dnp | YModuleProxy **get_module**( ) |
| java | YModule **get_module**( ) |
| py | **get_module**( ) |
| php | function **get_module**( ) |
| es | async **get_module**( ) |

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

**Returns :**
an instance of YModule

## gps→get_module_async() YGps
## gps→module_async()

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

```
js  function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as on-line.

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

**Parameters :**

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

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

**Returns :**

nothing : the result is provided to the callback.

## gps→get_satCount()
## gps→satCount()

YGps

Returns the total count of satellites used to compute GPS position.

| | |
|---|---|
| js | function **get_satCount**( ) |
| cpp | s64 **get_satCount**( ) |
| m | -(s64) satCount |
| pas | int64 **get_satCount**( ): int64 |
| vb | function **get_satCount**( ) As Long |
| cs | long **get_satCount**( ) |
| dnp | long **get_satCount**( ) |
| java | long **get_satCount**( ) |
| uwp | async Task<long> **get_satCount**( ) |
| py | **get_satCount**( ) |
| php | function **get_satCount**( ) |
| es | async **get_satCount**( ) |
| cmd | YGps **target get_satCount** |

**Returns :**
an integer corresponding to the total count of satellites used to compute GPS position

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

## gps→get_satPerConst() YGps
## gps→satPerConst()

Returns the count of visible satellites per constellation encoded on a 32 bit integer: bits 0..

```
js      function get_satPerConst( )
cpp     s64 get_satPerConst( )
m       -(s64) satPerConst
pas     int64 get_satPerConst( ): int64
vb      function get_satPerConst( ) As Long
cs      long get_satPerConst( )
dnp     long get_satPerConst( )
java    long get_satPerConst( )
uwp     async Task<long> get_satPerConst( )
py      get_satPerConst( )
php     function get_satPerConst( )
es      async get_satPerConst( )
cmd     YGps target get_satPerConst
```

5: GPS satellites count, bits 6..11 : Glonass, bits 12..17 : Galileo. this value is refreshed every 5 seconds only.

**Returns :**

an integer corresponding to the count of visible satellites per constellation encoded on a 32 bit integer: bits 0.

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

## gps→get_serialNumber() YGps
## gps→serialNumber()

Returns the serial number of the module, as set by the factory.

```
js      function get_serialNumber( )
cpp     string get_serialNumber( )
m       -(NSString*) serialNumber
pas     string get_serialNumber( ): string
vb      function get_serialNumber( ) As String
cs      string get_serialNumber( )
dnp     string get_serialNumber( )
java    String get_serialNumber( )
uwp     async Task<string> get_serialNumber( )
py      get_serialNumber( )
php     function get_serialNumber( )
es      async get_serialNumber( )
cmd     YGps 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 YModule.SERIALNUMBER_INVALID.

## gps→get_unixTime()
## gps→unixTime()

YGps

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

```
js      function get_unixTime( )
cpp     s64 get_unixTime( )
m       -(s64) unixTime
pas     int64 get_unixTime( ): int64
vb      function get_unixTime( ) As Long
cs      long get_unixTime( )
dnp     long get_unixTime( )
java    long get_unixTime( )
uwp     async Task<long> get_unixTime( )
py      get_unixTime( )
php     function get_unixTime( )
es      async get_unixTime( )
cmd     YGps target get_unixTime
```

**Returns :**
an integer corresponding to the current time in Unix format (number of seconds elapsed since Jan 1st, 1970)

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

## gps→get_userData()
## gps→userData()

YGps

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

| | |
|---|---|
| js | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(id) userData |
| pas | Tobject **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | **get_userData**( ) |
| php | function **get_userData**( ) |
| es | async **get_userData**( ) |

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

**Returns :**
the object stored previously by the caller.

## gps→get_utcOffset()
## gps→utcOffset()

YGps

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

| | |
|---|---|
| js | function **get_utcOffset**( ) |
| cpp | int **get_utcOffset**( ) |
| m | -(int) utcOffset |
| pas | LongInt **get_utcOffset**( ): LongInt |
| vb | function **get_utcOffset**( ) As Integer |
| cs | int **get_utcOffset**( ) |
| dnp | int **get_utcOffset**( ) |
| java | int **get_utcOffset**( ) |
| uwp | async Task<int> **get_utcOffset**( ) |
| py | **get_utcOffset**( ) |
| php | function **get_utcOffset**( ) |
| es | async **get_utcOffset**( ) |
| cmd | YGps **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`.

## gps→isOnline() YGps

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

| | |
|---|---|
| js | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | boolean **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| dnp | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | **isOnline**( ) |
| php | function **isOnline**( ) |
| es | async **isOnline**( ) |

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

**Returns :**
`true` if the geolocalization module can be reached, and `false` otherwise

## gps→isOnline_async() YGps

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

```
js  function isOnline_async( callback, context)
```

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

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

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

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

**Returns :**

nothing : the result is provided to the callback.

## gps→isReadOnly() YGps

Test if the function is readOnly.

| | |
|---|---|
| cpp | bool **isReadOnly**( ) |
| m | -(bool) **isReadOnly** |
| pas | boolean **isReadOnly**( ): boolean |
| vb | function **isReadOnly**( ) As Boolean |
| cs | bool **isReadOnly**( ) |
| dnp | bool **isReadOnly**( ) |
| java | boolean **isReadOnly**( ) |
| uwp | async Task<bool> **isReadOnly**( ) |
| py | **isReadOnly**( ) |
| php | function **isReadOnly**( ) |
| es | async **isReadOnly**( ) |
| cmd | YGps **target isReadOnly** |

Return `true` if the function is write protected or that the function is not available.

**Returns :**
`true` if the function is readOnly or not online.

## gps→load() YGps

Preloads the geolocalization module cache with a specified validity duration.

```
js      function load( msValidity)
cpp     YRETCODE load( int msValidity)
m       -(YRETCODE) load : (u64) msValidity
pas     YRETCODE load( msValidity: u64): YRETCODE
vb      function load( ByVal msValidity As Long) As YRETCODE
cs      YRETCODE load( ulong msValidity)
java    int load( long msValidity)
py      load( msValidity)
php     function load( $msValidity)
es      async load( msValidity)
```

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

**Parameters :**

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

**Returns :**

YAPI_SUCCESS when the call succeeds.

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

## gps→loadAttribute() YGps

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
js      function loadAttribute( attrName)
cpp     string loadAttribute( string attrName)
m       -(NSString*) loadAttribute : (NSString*) attrName
pas     string loadAttribute( attrName: string): string
vb      function loadAttribute( ) As String
cs      string loadAttribute( string attrName)
dnp     string loadAttribute( string attrName)
java    String loadAttribute( String attrName)
uwp     async Task<string> loadAttribute( string attrName)
py      loadAttribute( attrName)
php     function loadAttribute( $attrName)
es      async loadAttribute( attrName)
```

**Parameters :**

**attrName** the name of the requested attribute

**Returns :**

a string with the value of the the attribute

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

## gps→load_async() YGps

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

```
js  function load_async( msValidity, callback, context)
```

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

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking the JavaScript virtual machine.

**Parameters :**

| | |
|---|---|
| **msValidity** | an integer corresponding to the validity of the loaded function parameters, in milliseconds |
| **callback** | callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or YAPI_SUCCESS) |
| **context** | caller-specific object that is passed as-is to the callback function |

**Returns :**

nothing : the result is provided to the callback.

## gps→muteValueCallbacks() YGps

Disables the propagation of every new advertised value to the parent hub.

| | |
|---|---|
| js | function **muteValueCallbacks**( ) |
| cpp | int **muteValueCallbacks**( ) |
| m | -(int) **muteValueCallbacks** |
| pas | LongInt **muteValueCallbacks**( ): LongInt |
| vb | function **muteValueCallbacks**( ) As Integer |
| cs | int **muteValueCallbacks**( ) |
| dnp | int **muteValueCallbacks**( ) |
| java | int **muteValueCallbacks**( ) |
| uwp | async Task<int> **muteValueCallbacks**( ) |
| py | **muteValueCallbacks**( ) |
| php | function **muteValueCallbacks**( ) |
| es | async **muteValueCallbacks**( ) |
| cmd | YGps **target muteValueCallbacks** |

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Returns :**
`YAPI_SUCCESS` when the call succeeds.

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

## gps→nextGps() YGps

Continues the enumeration of geolocalization modules started using `yFirstGps()`.

```
js      function nextGps( )
cpp     YGps * nextGps( )
m       -(YGps*) nextGps
pas     TYGps nextGps( ): TYGps
vb      function nextGps( ) As YGps
cs      YGps nextGps( )
java    YGps nextGps( )
uwp     YGps nextGps( )
py      nextGps( )
php     function nextGps( )
es      nextGps( )
```

Caution: You can't make any assumption about the returned geolocalization modules order. If you want to find a specific a geolocalization module, use `Gps.findGps()` and a hardwareID or a logical name.

**Returns :**

a pointer to a `YGps` object, corresponding to a geolocalization module currently online, or a `null` pointer if there are no more geolocalization modules to enumerate.

## gps→registerValueCallback() YGps

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

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| cpp | int **registerValueCallback**( YGpsValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YGpsValueCallback) **callback** |
| pas | LongInt **registerValueCallback**( **callback**: TYGpsValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| uwp | async Task<int> **registerValueCallback**( ValueCallback **callback**) |
| py | **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| es | async **registerValueCallback**( **callback**) |

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

**Parameters :**

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

## gps→set_constellation()
## gps→setConstellation()

YGps

Changes the satellites constellation used to compute positioning data.

```
js      function set_constellation( newval)
cpp     int set_constellation( Y_CONSTELLATION_enum newval)
m       -(int) setConstellation : (Y_CONSTELLATION_enum) newval
pas     integer set_constellation( newval: Integer): integer
vb      function set_constellation( ByVal newval As Integer) As Integer
cs      int set_constellation( int newval)
dnp     int set_constellation( int newval)
java    int set_constellation( int newval)
uwp     async Task<int> set_constellation( int newval)
py      set_constellation( newval)
php     function set_constellation( $newval)
es      async set_constellation( newval)
cmd     YGps target set_constellation newval
```

Possible constellations are GNSS ( = all supported constellations), GPS, Glonass, Galileo , and the 3 possible pairs. This setting has no effect on Yocto-GPS (V1).

**Parameters :**

**newval** a value among `Y_CONSTELLATION_GNSS`, `Y_CONSTELLATION_GPS`, `Y_CONSTELLATION_GLONASS`, `Y_CONSTELLATION_GALILEO`, `Y_CONSTELLATION_GPS_GLONASS`, `Y_CONSTELLATION_GPS_GALILEO` and `Y_CONSTELLATION_GLONASS_GALILEO` corresponding to the satellites constellation used to compute positioning data

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## gps→set_coordSystem() YGps
## gps→setCoordSystem()

Changes the representation system used for positioning data.

```
js      function set_coordSystem( newval)
cpp     int set_coordSystem( Y_COORDSYSTEM_enum newval)
m       -(int) setCoordSystem : (Y_COORDSYSTEM_enum) newval
pas     integer set_coordSystem( newval: Integer): integer
vb      function set_coordSystem( ByVal newval As Integer) As Integer
cs      int set_coordSystem( int newval)
dnp     int set_coordSystem( int newval)
java    int set_coordSystem( int newval)
uwp     async Task<int> set_coordSystem( int newval)
py      set_coordSystem( newval)
php     function set_coordSystem( $newval)
es      async set_coordSystem( newval)
cmd     YGps target set_coordSystem newval
```

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

**Parameters :**

**newval** a value among `Y_COORDSYSTEM_GPS_DMS`, `Y_COORDSYSTEM_GPS_DM` and `Y_COORDSYSTEM_GPS_D` corresponding to the representation system used for positioning data

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## gps→set_logicalName() YGps
## gps→setLogicalName()

Changes the logical name of the geolocalization module.

```
js      function set_logicalName( newval)
cpp     int set_logicalName( string newval)
m       -(int) setLogicalName : (NSString*) newval
pas     integer set_logicalName( newval: string): integer
vb      function set_logicalName( ByVal newval As String) As Integer
cs      int set_logicalName( string newval)
dnp     int set_logicalName( string newval)
java    int set_logicalName( String newval)
uwp     async Task<int> set_logicalName( string newval)
py      set_logicalName( newval)
php     function set_logicalName( $newval)
es      async set_logicalName( newval)
cmd     YGps 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 geolocalization module.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## gps→set_userData()
## gps→setUserData()

YGps

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

```
js      function set_userData( data)
cpp     void set_userData( void * data)
m       -(void) setUserData : (id) data
pas     set_userData( data: Tobject)
vb      procedure set_userData( ByVal data As Object)
cs      void set_userData( object data)
java    void set_userData( Object data)
py      set_userData( data)
php     function set_userData( $data)
es      async set_userData( data)
```

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

**Parameters :**

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

## gps→set_utcOffset()
## gps→setUtcOffset()

YGps

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

```
js      function set_utcOffset( newval)
cpp     int set_utcOffset( int newval)
m       -(int) setUtcOffset : (int) newval
pas     integer set_utcOffset( newval: LongInt): integer
vb      function set_utcOffset( ByVal newval As Integer) As Integer
cs      int set_utcOffset( int newval)
dnp     int set_utcOffset( int newval)
java    int set_utcOffset( int newval)
uwp     async Task<int> set_utcOffset( int newval)
py      set_utcOffset( newval)
php     function set_utcOffset( $newval)
es      async set_utcOffset( newval)
cmd     YGps 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 is automatically be updated according to the selected time zone. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**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.

## gps→unmuteValueCallbacks() YGps

Re-enables the propagation of every new advertised value to the parent hub.

| | |
|---|---|
| js | function **unmuteValueCallbacks**( ) |
| cpp | int **unmuteValueCallbacks**( ) |
| m | -(int) **unmuteValueCallbacks** |
| pas | LongInt **unmuteValueCallbacks**( ): LongInt |
| vb | function **unmuteValueCallbacks**( ) As Integer |
| cs | int **unmuteValueCallbacks**( ) |
| dnp | int **unmuteValueCallbacks**( ) |
| java | int **unmuteValueCallbacks**( ) |
| uwp | async Task<int> **unmuteValueCallbacks**( ) |
| py | **unmuteValueCallbacks**( ) |
| php | function **unmuteValueCallbacks**( ) |
| es | async **unmuteValueCallbacks**( ) |
| cmd | YGps **target unmuteValueCallbacks** |

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Returns :**
`YAPI_SUCCESS` when the call succeeds.

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

## gps→wait_async() YGps

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

```
js  function wait_async( callback, context)
es  wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

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

**Returns :**

nothing.

# 22.4. Class YLatitude

Latitude sensor control interface, available for instance in the Yocto-GPS

The `YLatitude` class allows you to read and configure Yoctopuce latitude sensors. It inherits from `YSensor` class the core functions to read measurements, to register callback functions, and to access the autonomous datalogger.

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

| | |
|---|---|
| es | in HTML: <script src="../../lib/yocto_latitude.js"></script><br>in node.js: require('yoctolib-es2017/yocto_latitude.js'); |
| js | <script type='text/javascript' src='yocto_latitude.js'></script> |
| cpp | #include "yocto_latitude.h" |
| m | #import "yocto_latitude.h" |
| pas | uses yocto_latitude; |
| vb | yocto_latitude.vb |
| cs | yocto_latitude.cs |
| dnp | import YoctoProxyAPI.YLatitudeProxy |
| java | import com.yoctopuce.YoctoAPI.YLatitude; |
| uwp | import com.yoctopuce.YoctoAPI.YLatitude; |
| py | from yocto_latitude import * |
| php | require_once('yocto_latitude.php'); |
| vi | YLatitude.vi |

| Global functions | |
|---|---|
| **YLatitude.FindLatitude**(**func**) | Retrieves a latitude sensor for a given identifier. |
| **YLatitude.FindLatitudeInContext**(**yctx**, **func**) | Retrieves a latitude sensor for a given identifier in a YAPI context. |
| **YLatitude.FirstLatitude**() | Starts the enumeration of latitude sensors currently accessible. |
| **YLatitude.FirstLatitudeInContext**(**yctx**) | Starts the enumeration of latitude sensors currently accessible. |
| **YLatitude.GetSimilarFunctions**() | Enumerates all functions of type Latitude available on the devices currently reachable by the library, and returns their unique hardware ID. |

| `YLatitude` properties | |
|---|---|
| **latitude→AdvMode** *[writable]* | Measuring mode used for the advertised value pushed to the parent hub. |
| **latitude→AdvertisedValue** *[read-only]* | Short string representing the current state of the function. |
| **latitude→FriendlyName** *[read-only]* | Global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`. |
| **latitude→FunctionId** *[read-only]* | Hardware identifier of the sensor, without reference to the module. |
| **latitude→HardwareId** *[read-only]* | Unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`. |

**latitude→IsOnline** *[read-only]*

Checks if the function is currently reachable.

**latitude→LogFrequency** *[writable]*

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

**latitude→LogicalName** *[writable]*

Logical name of the function.

**latitude→ReportFrequency** *[writable]*

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

**latitude→Resolution** *[writable]*

Resolution of the measured values.

**latitude→SerialNumber** *[read-only]*

Serial number of the module, as set by the factory.

### `YLatitude` methods

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

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

**latitude→clearCache()**

Invalidates the cache.

**latitude→describe()**

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

**latitude→get_advMode()**

Returns the measuring mode used for the advertised value pushed to the parent hub.

**latitude→get_advertisedValue()**

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

**latitude→get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor, in deg/1000, as a floating point number.

**latitude→get_currentValue()**

Returns the current value of the latitude, in deg/1000, as a floating point number.

**latitude→get_dataLogger()**

Returns the `YDatalogger` object of the device hosting the sensor.

**latitude→get_errorMessage()**

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

**latitude→get_errorType()**

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

**latitude→get_friendlyName()**

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

**latitude→get_functionDescriptor()**

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

**latitude→get_functionId()**

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

**latitude→get_hardwareId()**

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

**latitude→get_highestValue()**

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

**latitude→get_logFrequency**()

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

**latitude→get_logicalName**()

Returns the logical name of the latitude sensor.

**latitude→get_lowestValue**()

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

**latitude→get_module**()

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

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

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

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

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

**latitude→get_reportFrequency**()

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

**latitude→get_resolution**()

Returns the resolution of the measured values.

**latitude→get_sensorState**()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

**latitude→get_serialNumber**()

Returns the serial number of the module, as set by the factory.

**latitude→get_unit**()

Returns the measuring unit for the latitude.

**latitude→get_userData**()

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

**latitude→isOnline**()

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

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

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

**latitude→isReadOnly**()

Test if the function is readOnly.

**latitude→isSensorReady**()

Checks if the sensor is currently able to provide an up-to-date measure.

**latitude→load**(**msValidity**)

Preloads the latitude sensor cache with a specified validity duration.

**latitude→loadAttribute**(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

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

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

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

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

**latitude→muteValueCallbacks**()

Disables the propagation of every new advertised value to the parent hub.

**latitude→nextLatitude**()

Continues the enumeration of latitude sensors started using `yFirstLatitude()`.

**latitude→registerTimedReportCallback**(**callback**)

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

**latitude→registerValueCallback**(**callback**)

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

**latitude→set_advMode**(**newval**)

Changes the measuring mode used for the advertised value pushed to the parent hub.

**latitude→set_highestValue**(**newval**)

Changes the recorded maximal value observed.

**latitude→set_logFrequency**(**newval**)

Changes the datalogger recording frequency for this function.

**latitude→set_logicalName**(**newval**)

Changes the logical name of the latitude sensor.

**latitude→set_lowestValue**(**newval**)

Changes the recorded minimal value observed.

**latitude→set_reportFrequency**(**newval**)

Changes the timed value notification frequency for this function.

**latitude→set_resolution**(**newval**)

Changes the resolution of the measured physical values.

**latitude→set_userData**(**data**)

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

**latitude→startDataLogger**()

Starts the data logger on the device.

**latitude→stopDataLogger**()

Stops the datalogger on the device.

**latitude→unmuteValueCallbacks**()

Re-enables the propagation of every new advertised value to the parent hub.

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

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

## YLatitude.FindLatitude() YLatitude
## YLatitude.FindLatitude()

Retrieves a latitude sensor for a given identifier.

```
js      function yFindLatitude( func)
cpp     YLatitude* yFindLatitude( string func)
m       +(YLatitude*) FindLatitude : (NSString*) func
pas     TYLatitude yFindLatitude( func: string): TYLatitude
vb      function yFindLatitude( ByVal func As String) As YLatitude
cs      static YLatitude FindLatitude( string func)
dnp     static YLatitudeProxy FindLatitude( string func)
java    static YLatitude FindLatitude( String func)
uwp     static YLatitude FindLatitude( string func)
py      FindLatitude( func)
php     function yFindLatitude( $func)
es      static FindLatitude( func)
```

The identifier can be specified using several formats:

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

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

If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

**Parameters :**

**func** a string that uniquely characterizes the latitude sensor, for instance `YGNSSMK1.latitude`.

**Returns :**

a `YLatitude` object allowing you to drive the latitude sensor.

## YLatitude.FindLatitudeInContext() YLatitude YLatitude.FindLatitudeInContext()

Retrieves a latitude sensor for a given identifier in a YAPI context.

java static YLatitude **FindLatitudeInContext**( YAPIContext **yctx**, String **func**)
uwp static YLatitude **FindLatitudeInContext**( YAPIContext **yctx**, string **func**)
es static **FindLatitudeInContext**( **yctx**, **func**)

The identifier can be specified using several formats:

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

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

**Parameters :**

**yctx** a YAPI context

**func** a string that uniquely characterizes the latitude sensor, for instance `YGNSSMK1.latitude`.

**Returns :**

a `YLatitude` object allowing you to drive the latitude sensor.

## YLatitude.FirstLatitude() YLatitude
## YLatitude.FirstLatitude()

Starts the enumeration of latitude sensors currently accessible.

| | |
|---|---|
| js | function **yFirstLatitude**( ) |
| cpp | YLatitude * **yFirstLatitude**( ) |
| m | +(YLatitude*) **FirstLatitude** |
| pas | TYLatitude **yFirstLatitude**( ): TYLatitude |
| vb | function **yFirstLatitude**( ) As YLatitude |
| cs | static YLatitude **FirstLatitude**( ) |
| java | static YLatitude **FirstLatitude**( ) |
| uwp | static YLatitude **FirstLatitude**( ) |
| py | **FirstLatitude**( ) |
| php | function **yFirstLatitude**( ) |
| es | static **FirstLatitude**( ) |

Use the method `YLatitude.nextLatitude()` to iterate on next latitude sensors.

**Returns :**

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

## YLatitude.FirstLatitudeInContext() YLatitude
## YLatitude.FirstLatitudeInContext()

Starts the enumeration of latitude sensors currently accessible.

```
java  static YLatitude FirstLatitudeInContext( YAPIContext yctx)
uwp   static YLatitude FirstLatitudeInContext( YAPIContext yctx)
es    static FirstLatitudeInContext( yctx)
```

Use the method `YLatitude.nextLatitude()` to iterate on next latitude sensors.

**Parameters :**

**yctx** a YAPI context.

**Returns :**

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

## YLatitude.GetSimilarFunctions() YLatitude YLatitude.GetSimilarFunctions()

Enumerates all functions of type Latitude available on the devices currently reachable by the library, and returns their unique hardware ID.

dnp static new string[] **GetSimilarFunctions**( )

Each of these IDs can be provided as argument to the method `YLatitude.FindLatitude` to obtain an object that can control the corresponding device.

**Returns :**
an array of strings, each string containing the unique hardwareId of a device function currently connected.

## latitude→AdvMode YLatitude

Measuring mode used for the advertised value pushed to the parent hub.

dnp int **AdvMode**

**Possible values:**

Y_ADVMODE_INVALID = 0
Y_ADVMODE_IMMEDIATE = 1
Y_ADVMODE_PERIOD_AVG = 2
Y_ADVMODE_PERIOD_MIN = 3
Y_ADVMODE_PERIOD_MAX = 4

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

## latitude→AdvertisedValue — YLatitude

Short string representing the current state of the function.

dnp string **AdvertisedValue**

## latitude→FriendlyName — YLatitude

Global identifier of the function in the format MODULE_NAME.FUNCTION_NAME.

dnp string **FriendlyName**

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

## latitude→FunctionId — YLatitude

Hardware identifier of the sensor, without reference to the module.

dnp string **FunctionId**

For example `relay1`

## latitude→HardwareId — YLatitude

Unique hardware identifier of the function in the form SERIAL.FUNCTIONID.

dnp string **HardwareId**

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

## latitude→IsOnline — YLatitude

Checks if the function is currently reachable.

dnp bool **IsOnline**

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

## latitude→LogFrequency — YLatitude

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

dnp string **LogFrequency**

**Writable**. Changes the datalogger recording frequency for this function. 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". Note that setting the datalogger recording frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

## latitude→LogicalName — YLatitude

Logical name of the function.

dnp string **LogicalName**

**Writable**. 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.

## latitude→ReportFrequency — YLatitude

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

dnp string **ReportFrequency**

**Writable**. Changes the timed value notification frequency for this function. The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (e.g. "4/h"). To disable timed value notifications for this function, use the value "OFF". Note that setting the timed value notification frequency to a greater value than the sensor native sampling frequency is unless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

## latitude→Resolution — YLatitude

Resolution of the measured values.

dnp double **Resolution**

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Writable**. Changes the resolution of the measured physical values. The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

## latitude→SerialNumber — YLatitude

Serial number of the module, as set by the factory.

dnp string **SerialNumber**

## latitude→calibrateFromPoints() YLatitude

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

```
js    function calibrateFromPoints( rawValues, refValues)
cpp   int calibrateFromPoints( vector<double> rawValues,
                               vector<double> refValues)
m     -(int) calibrateFromPoints : (NSMutableArray*) rawValues
                                 : (NSMutableArray*) refValues
pas   LongInt calibrateFromPoints( rawValues: TDoubleArray,
                                   refValues: TDoubleArray): LongInt
vb    procedure calibrateFromPoints( )
cs    int calibrateFromPoints( List<double> rawValues,
                               List<double> refValues)
dnp   int calibrateFromPoints( )
java  int calibrateFromPoints( ArrayList<Double> rawValues,
                               ArrayList<Double> refValues)
uwp   async Task<int> calibrateFromPoints( List<double> rawValues,
                                           List<double> refValues)
py    calibrateFromPoints( rawValues, refValues)
php   function calibrateFromPoints( $rawValues, $refValues)
es    async calibrateFromPoints( rawValues, refValues)
cmd   YLatitude 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.

## latitude→clearCache() YLatitude

Invalidates the cache.

| | |
|---|---|
| js | function **clearCache**( ) |
| cpp | void **clearCache**( ) |
| m | -(void) **clearCache** |
| pas | **clearCache**( ) |
| vb | procedure **clearCache**( ) |
| cs | void **clearCache**( ) |
| java | void **clearCache**( ) |
| py | **clearCache**( ) |
| php | function **clearCache**( ) |
| es | async **clearCache**( ) |

Invalidates the cache of the latitude sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

## latitude→describe() YLatitude

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

```
js      function describe( )
cpp     string describe( )
m       -(NSString*) describe
pas     string describe( ): string
vb      function describe( ) As String
cs      string describe( )
java    String describe( )
py      describe( )
php     function describe( )
es      async describe( )
```

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

**Returns :**
a string that describes the latitude sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

## latitude→get_advMode()
## latitude→advMode()

YLatitude

Returns the measuring mode used for the advertised value pushed to the parent hub.

| | |
|---|---|
| js | function **get_advMode**( ) |
| cpp | Y_ADVMODE_enum **get_advMode**( ) |
| m | -(Y_ADVMODE_enum) advMode |
| pas | Integer **get_advMode**( ): Integer |
| vb | function **get_advMode**( ) As Integer |
| cs | int **get_advMode**( ) |
| dnp | int **get_advMode**( ) |
| java | int **get_advMode**( ) |
| uwp | async Task<int> **get_advMode**( ) |
| py | **get_advMode**( ) |
| php | function **get_advMode**( ) |
| es | async **get_advMode**( ) |
| cmd | YLatitude **target get_advMode** |

**Returns :**

a value among `Y_ADVMODE_IMMEDIATE`, `Y_ADVMODE_PERIOD_AVG`, `Y_ADVMODE_PERIOD_MIN` and `Y_ADVMODE_PERIOD_MAX` corresponding to the measuring mode used for the advertised value pushed to the parent hub

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

## latitude→get_advertisedValue() YLatitude
## latitude→advertisedValue()

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

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | string **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| dnp | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| uwp | async Task<string> **get_advertisedValue**( ) |
| py | **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| es | async **get_advertisedValue**( ) |
| cmd | YLatitude **target get_advertisedValue** |

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

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

## latitude→get_currentRawValue() YLatitude
## latitude→currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in deg/1000, as a floating point number.

```
js      function get_currentRawValue( )
cpp     double get_currentRawValue( )
m       -(double) currentRawValue
pas     double get_currentRawValue( ): double
vb      function get_currentRawValue( ) As Double
cs      double get_currentRawValue( )
dnp     double get_currentRawValue( )
java    double get_currentRawValue( )
uwp     async Task<double> get_currentRawValue( )
py      get_currentRawValue( )
php     function get_currentRawValue( )
es      async get_currentRawValue( )
cmd     YLatitude target get_currentRawValue
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in deg/1000, as a floating point number

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

## latitude→get_currentValue() YLatitude
## latitude→currentValue()

Returns the current value of the latitude, in deg/1000, as a floating point number.

| | |
|---|---|
| js | function **get_currentValue**( ) |
| cpp | double **get_currentValue**( ) |
| m | -(double) currentValue |
| pas | double **get_currentValue**( ): double |
| vb | function **get_currentValue**( ) As Double |
| cs | double **get_currentValue**( ) |
| dnp | double **get_currentValue**( ) |
| java | double **get_currentValue**( ) |
| uwp | async Task<double> **get_currentValue**( ) |
| py | **get_currentValue**( ) |
| php | function **get_currentValue**( ) |
| es | async **get_currentValue**( ) |
| cmd | YLatitude **target get_currentValue** |

Note that a get_currentValue() call will *not* start a measure in the device, it will just return the last measure that occurred in the device. Indeed, internally, each Yoctopuce devices is continuously making measurements at a hardware specific frequency.

If continuously calling get_currentValue() leads you to performances issues, then you might consider to switch to callback programming model. Check the "advanced programming" chapter in in your device user manual for more information.

**Returns :**

a floating point number corresponding to the current value of the latitude, in deg/1000, as a floating point number

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

## latitude→get_dataLogger()
## latitude→dataLogger()

**YLatitude**

Returns the `YDatalogger` object of the device hosting the sensor.

| | |
|---|---|
| js | function **get_dataLogger**( ) |
| cpp | YDataLogger* **get_dataLogger**( ) |
| m | -(YDataLogger*) dataLogger |
| pas | TYDataLogger **get_dataLogger**( ): TYDataLogger |
| vb | function **get_dataLogger**( ) As YDataLogger |
| cs | YDataLogger **get_dataLogger**( ) |
| dnp | YDataLoggerProxy **get_dataLogger**( ) |
| java | YDataLogger **get_dataLogger**( ) |
| uwp | async Task<YDataLogger> **get_dataLogger**( ) |
| py | **get_dataLogger**( ) |
| php | function **get_dataLogger**( ) |
| es | async **get_dataLogger**( ) |

This method returns an object that can control global parameters of the data logger. The returned object should not be freed.

**Returns :**
an `YDatalogger` object, or null on error.

## latitude→get_errorMessage() YLatitude
## latitude→errorMessage()

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

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | string **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| es | **get_errorMessage**( ) |

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

**Returns :**
a string corresponding to the latest error message that occured while using the latitude sensor object

## latitude→get_errorType() YLatitude
## latitude→errorType()

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

| | |
|---|---|
| js | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| m | -(YRETCODE) errorType |
| pas | YRETCODE **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | **get_errorType**( ) |
| php | function **get_errorType**( ) |
| es | **get_errorType**( ) |

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

**Returns :**
a number corresponding to the code of the latest error that occurred while using the latitude sensor object

## latitude→get_friendlyName() YLatitude
## latitude→friendlyName()

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

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| dnp | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| es | async **get_friendlyName**( ) |

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

**Returns :**

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

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

## latitude→get_functionDescriptor() YLatitude
## latitude→functionDescriptor()

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

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | YFUN_DESCR **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| es | async **get_functionDescriptor**( ) |

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

**Returns :**
an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

## latitude→get_functionId()
## latitude→functionId()

YLatitude

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

| | |
|---|---|
| js | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| dnp | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | **get_functionId**( ) |
| php | function **get_functionId**( ) |
| es | async **get_functionId**( ) |

For example `relay1`

**Returns :**
a string that identifies the latitude sensor (ex: `relay1`)

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

## latitude→get_hardwareId()
## latitude→hardwareId()

YLatitude

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

```
js      function get_hardwareId( )
cpp     string get_hardwareId( )
m       -(NSString*) hardwareId
vb      function get_hardwareId( ) As String
cs      string get_hardwareId( )
dnp     string get_hardwareId( )
java    String get_hardwareId( )
py      get_hardwareId( )
php     function get_hardwareId( )
es      async get_hardwareId( )
```

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

**Returns :**
a string that uniquely identifies the latitude sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

## latitude→get_highestValue() YLatitude
## latitude→highestValue()

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

```
js      function get_highestValue( )
cpp     double get_highestValue( )
m       -(double) highestValue
pas     double get_highestValue( ): double
vb      function get_highestValue( ) As Double
cs      double get_highestValue( )
dnp     double get_highestValue( )
java    double get_highestValue( )
uwp     async Task<double> get_highestValue( )
py      get_highestValue( )
php     function get_highestValue( )
es      async get_highestValue( )
cmd     YLatitude target get_highestValue
```

Can be reset to an arbitrary value thanks to set_highestValue().

**Returns :**

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

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

## latitude→get_logFrequency()
## latitude→logFrequency()

YLatitude

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

```
js      function get_logFrequency( )
cpp     string get_logFrequency( )
m       -(NSString*) logFrequency
pas     string get_logFrequency( ): string
vb      function get_logFrequency( ) As String
cs      string get_logFrequency( )
dnp     string get_logFrequency( )
java    String get_logFrequency( )
uwp     async Task<string> get_logFrequency( )
py      get_logFrequency( )
php     function get_logFrequency( )
es      async get_logFrequency( )
cmd     YLatitude 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`.

## latitude→get_logicalName()
## latitude→logicalName()

YLatitude

Returns the logical name of the latitude sensor.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | string **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| dnp | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| uwp | async Task<string> **get_logicalName**( ) |
| py | **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| es | async **get_logicalName**( ) |
| cmd | YLatitude **target get_logicalName** |

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

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

## latitude→get_lowestValue()
## latitude→lowestValue()

YLatitude

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

```
js      function get_lowestValue( )
cpp     double get_lowestValue( )
m       -(double) lowestValue
pas     double get_lowestValue( ): double
vb      function get_lowestValue( ) As Double
cs      double get_lowestValue( )
dnp     double get_lowestValue( )
java    double get_lowestValue( )
uwp     async Task<double> get_lowestValue( )
py      get_lowestValue( )
php     function get_lowestValue( )
es      async get_lowestValue( )
cmd     YLatitude target get_lowestValue
```

Can be reset to an arbitrary value thanks to set_lowestValue().

**Returns :**

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

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

## latitude→get_module()
## latitude→module()

**YLatitude**

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

| | |
|---|---|
| js | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | TYModule **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| dnp | YModuleProxy **get_module**( ) |
| java | YModule **get_module**( ) |
| py | **get_module**( ) |
| php | function **get_module**( ) |
| es | async **get_module**( ) |

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

**Returns :**
an instance of YModule

## latitude→get_module_async() YLatitude
## latitude→module_async()

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

```
js  function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as online.

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

**Parameters :**

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

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

**Returns :**

nothing : the result is provided to the callback.

## latitude→get_recordedData() YLatitude
## latitude→recordedData()

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

```
js      function get_recordedData( startTime, endTime)
cpp     YDataSet get_recordedData( double startTime, double endTime)
m       -(YDataSet*) recordedData : (double) startTime
                                  : (double) endTime
pas     TYDataSet get_recordedData( startTime: double, endTime: double): TYDataSet
vb      function get_recordedData( ) As YDataSet
cs      YDataSet get_recordedData( double startTime, double endTime)
dnp     YDataSetProxy get_recordedData( double startTime,
                                        double endTime)
java    YDataSet get_recordedData( double startTime, double endTime)
uwp     async Task<YDataSet> get_recordedData( double startTime,
                                               double endTime)
py      get_recordedData( startTime, endTime)
php     function get_recordedData( $startTime, $endTime)
es      async get_recordedData( startTime, endTime)
cmd     YLatitude 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 YDataSet 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 YDataSet 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 measure, 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 measure, 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.

## latitude→get_reportFrequency()
## latitude→reportFrequency()

YLatitude

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

```
js      function get_reportFrequency( )
cpp     string get_reportFrequency( )
m       -(NSString*) reportFrequency
pas     string get_reportFrequency( ): string
vb      function get_reportFrequency( ) As String
cs      string get_reportFrequency( )
dnp     string get_reportFrequency( )
java    String get_reportFrequency( )
uwp     async Task<string> get_reportFrequency( )
py      get_reportFrequency( )
php     function get_reportFrequency( )
es      async get_reportFrequency( )
cmd     YLatitude 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`.

## latitude→get_resolution()
## latitude→resolution()

YLatitude

Returns the resolution of the measured values.

```
js      function get_resolution( )
cpp     double get_resolution( )
m       -(double) resolution
pas     double get_resolution( ): double
vb      function get_resolution( ) As Double
cs      double get_resolution( )
dnp     double get_resolution( )
java    double get_resolution( )
uwp     async Task<double> get_resolution( )
py      get_resolution( )
php     function get_resolution( )
es      async get_resolution( )
cmd     YLatitude 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. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

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

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

## latitude→get_sensorState()
## latitude→sensorState()

**YLatitude**

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

```
js      function get_sensorState( )
cpp     int get_sensorState( )
m       -(int) sensorState
pas     LongInt get_sensorState( ): LongInt
vb      function get_sensorState( ) As Integer
cs      int get_sensorState( )
dnp     int get_sensorState( )
java    int get_sensorState( )
uwp     async Task<int> get_sensorState( )
py      get_sensorState( )
php     function get_sensorState( )
es      async get_sensorState( )
cmd     YLatitude target get_sensorState
```

**Returns :**

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns Y_SENSORSTATE_INVALID.

## latitude→get_serialNumber() YLatitude
## latitude→serialNumber()

Returns the serial number of the module, as set by the factory.

```
js      function get_serialNumber( )
cpp     string get_serialNumber( )
m       -(NSString*) serialNumber
pas     string get_serialNumber( ): string
vb      function get_serialNumber( ) As String
cs      string get_serialNumber( )
dnp     string get_serialNumber( )
java    String get_serialNumber( )
uwp     async Task<string> get_serialNumber( )
py      get_serialNumber( )
php     function get_serialNumber( )
es      async get_serialNumber( )
cmd     YLatitude 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 YModule.SERIALNUMBER_INVALID.

## latitude→get_unit()
## latitude→unit()

YLatitude

Returns the measuring unit for the latitude.

```
js    function get_unit( )
cpp   string get_unit( )
m     -(NSString*) unit
pas   string get_unit( ): string
vb    function get_unit( ) As String
cs    string get_unit( )
dnp   string get_unit( )
java  String get_unit( )
uwp   async Task<string> get_unit( )
py    get_unit( )
php   function get_unit( )
es    async get_unit( )
cmd   YLatitude target get_unit
```

**Returns :**
a string corresponding to the measuring unit for the latitude

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

## latitude→get_userData()
## latitude→userData()

YLatitude

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

| | |
|---|---|
| js | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(id) userData |
| pas | Tobject **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | **get_userData**( ) |
| php | function **get_userData**( ) |
| es | async **get_userData**( ) |

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

**Returns :**
the object stored previously by the caller.

## latitude→isOnline() YLatitude

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

```
js      function isOnline( )
cpp     bool isOnline( )
m       -(BOOL) isOnline
pas     boolean isOnline( ): boolean
vb      function isOnline( ) As Boolean
cs      bool isOnline( )
dnp     bool isOnline( )
java    boolean isOnline( )
py      isOnline( )
php     function isOnline( )
es      async isOnline( )
```

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

**Returns :**
`true` if the latitude sensor can be reached, and `false` otherwise

## latitude→isOnline_async() YLatitude

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

```
js  function isOnline_async( callback, context)
```

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

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

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

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

**Returns :**

nothing : the result is provided to the callback.

## latitude→isReadOnly() YLatitude

Test if the function is readOnly.

| | |
|---|---|
| cpp | bool **isReadOnly**( ) |
| m | -(bool) **isReadOnly** |
| pas | boolean **isReadOnly**( ): boolean |
| vb | function **isReadOnly**( ) As Boolean |
| cs | bool **isReadOnly**( ) |
| dnp | bool **isReadOnly**( ) |
| java | boolean **isReadOnly**( ) |
| uwp | async Task<bool> **isReadOnly**( ) |
| py | **isReadOnly**( ) |
| php | function **isReadOnly**( ) |
| es | async **isReadOnly**( ) |
| cmd | YLatitude **target isReadOnly** |

Return `true` if the function is write protected or that the function is not available.

**Returns :**
`true` if the function is readOnly or not online.

## latitude→isSensorReady() YLatitude

Checks if the sensor is currently able to provide an up-to-date measure.

cmd YLatitude **target isSensorReady**

Returns false if the device is unreachable, or if the sensor does not have a current measure to transmit. No exception is raised if there is an error while trying to contact the device hosting $THEFUNCTION$.

**Returns :**
`true` if the sensor can provide an up-to-date measure, and `false` otherwise

## latitude→load() YLatitude

Preloads the latitude sensor cache with a specified validity duration.

```
js      function load( msValidity)
cpp     YRETCODE load( int msValidity)
m       -(YRETCODE) load : (u64) msValidity
pas     YRETCODE load( msValidity: u64): YRETCODE
vb      function load( ByVal msValidity As Long) As YRETCODE
cs      YRETCODE load( ulong msValidity)
java    int load( long msValidity)
py      load( msValidity)
php     function load( $msValidity)
es      async load( msValidity)
```

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

**Parameters :**

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

**Returns :**

YAPI_SUCCESS when the call succeeds.

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

## latitude→loadAttribute() YLatitude

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
js      function loadAttribute( attrName)
cpp     string loadAttribute( string attrName)
m       -(NSString*) loadAttribute : (NSString*) attrName
pas     string loadAttribute( attrName: string): string
vb      function loadAttribute( ) As String
cs      string loadAttribute( string attrName)
dnp     string loadAttribute( string attrName)
java    String loadAttribute( String attrName)
uwp     async Task<string> loadAttribute( string attrName)
py      loadAttribute( attrName)
php     function loadAttribute( $attrName)
es      async loadAttribute( attrName)
```

**Parameters :**

**attrName** the name of the requested attribute

**Returns :**

a string with the value of the the attribute

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

## latitude→loadCalibrationPoints() YLatitude

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

```
js    function loadCalibrationPoints( rawValues, refValues)
cpp   int loadCalibrationPoints( vector<double> rawValues,
                                 vector<double> refValues)
m     -(int) loadCalibrationPoints : (NSMutableArray*) rawValues
                                   : (NSMutableArray*) refValues
pas   LongInt loadCalibrationPoints( var rawValues: TDoubleArray,
                                     var refValues: TDoubleArray): LongInt
vb    procedure loadCalibrationPoints( )
cs    int loadCalibrationPoints( List<double> rawValues,
                                 List<double> refValues)
dnp   int loadCalibrationPoints( )
java  int loadCalibrationPoints( ArrayList<Double> rawValues,
                                 ArrayList<Double> refValues)
uwp   async Task<int> loadCalibrationPoints( List<double> rawValues,
                                             List<double> refValues)
py    loadCalibrationPoints( rawValues, refValues)
php   function loadCalibrationPoints( &$rawValues, &$refValues)
es    async loadCalibrationPoints( rawValues, refValues)
cmd   YLatitude 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.

## latitude→load_async() YLatitude

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

```
js  function load_async( msValidity, callback, context)
```

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

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking the JavaScript virtual machine.

**Parameters :**

| | |
|---|---|
| **msValidity** | an integer corresponding to the validity of the loaded function parameters, in milliseconds |
| **callback** | callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or YAPI_SUCCESS) |
| **context** | caller-specific object that is passed as-is to the callback function |

**Returns :**

nothing : the result is provided to the callback.

## latitude→muteValueCallbacks() YLatitude

Disables the propagation of every new advertised value to the parent hub.

| | |
|---|---|
| js | function **muteValueCallbacks**( ) |
| cpp | int **muteValueCallbacks**( ) |
| m | -(int) **muteValueCallbacks** |
| pas | LongInt **muteValueCallbacks**( ): LongInt |
| vb | function **muteValueCallbacks**( ) As Integer |
| cs | int **muteValueCallbacks**( ) |
| dnp | int **muteValueCallbacks**( ) |
| java | int **muteValueCallbacks**( ) |
| uwp | async Task<int> **muteValueCallbacks**( ) |
| py | **muteValueCallbacks**( ) |
| php | function **muteValueCallbacks**( ) |
| es | async **muteValueCallbacks**( ) |
| cmd | YLatitude **target muteValueCallbacks** |

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Returns :**
`YAPI_SUCCESS` when the call succeeds.

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

## latitude→nextLatitude() YLatitude

Continues the enumeration of latitude sensors started using `yFirstLatitude()`.

| | |
|---|---|
| `js` | function **nextLatitude**( ) |
| `cpp` | YLatitude * **nextLatitude**( ) |
| `m` | -(YLatitude*) **nextLatitude** |
| `pas` | TYLatitude **nextLatitude**( ): TYLatitude |
| `vb` | function **nextLatitude**( ) As YLatitude |
| `cs` | YLatitude **nextLatitude**( ) |
| `java` | YLatitude **nextLatitude**( ) |
| `uwp` | YLatitude **nextLatitude**( ) |
| `py` | **nextLatitude**( ) |
| `php` | function **nextLatitude**( ) |
| `es` | **nextLatitude**( ) |

Caution: You can't make any assumption about the returned latitude sensors order. If you want to find a specific a latitude sensor, use `Latitude.findLatitude()` and a hardwareID or a logical name.

**Returns :**

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

## latitude→registerTimedReportCallback() YLatitude

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

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| cpp | int **registerTimedReportCallback**( YLatitudeTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YLatitudeTimedReportCallback) **callback** |
| pas | LongInt **registerTimedReportCallback**( **callback**: TYLatitudeTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| uwp | async Task<int> **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| es | async **registerTimedReportCallback**( **callback**) |

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

**Parameters :**

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

## latitude→registerValueCallback() YLatitude

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

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| cpp | int **registerValueCallback**( YLatitudeValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YLatitudeValueCallback) **callback** |
| pas | LongInt **registerValueCallback**( **callback**: TYLatitudeValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| uwp | async Task<int> **registerValueCallback**( ValueCallback **callback**) |
| py | **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| es | async **registerValueCallback**( **callback**) |

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

**Parameters :**

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

## latitude→set_advMode() YLatitude
## latitude→setAdvMode()

Changes the measuring mode used for the advertised value pushed to the parent hub.

```
js      function set_advMode( newval)
cpp     int set_advMode( Y_ADVMODE_enum newval)
m       -(int) setAdvMode : (Y_ADVMODE_enum) newval
pas     integer set_advMode( newval: Integer): integer
vb      function set_advMode( ByVal newval As Integer) As Integer
cs      int set_advMode( int newval)
dnp     int set_advMode( int newval)
java    int set_advMode( int newval)
uwp     async Task<int> set_advMode( int newval)
py      set_advMode( newval)
php     function set_advMode( $newval)
es      async set_advMode( newval)
cmd     YLatitude target set_advMode newval
```

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

**Parameters :**

**newval** a value among `Y_ADVMODE_IMMEDIATE`, `Y_ADVMODE_PERIOD_AVG`, `Y_ADVMODE_PERIOD_MIN` and `Y_ADVMODE_PERIOD_MAX` corresponding to the measuring mode used for the advertised value pushed to the parent hub

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## latitude→set_highestValue() YLatitude
## latitude→setHighestValue()

Changes the recorded maximal value observed.

```
js    function set_highestValue( newval)
cpp   int set_highestValue( double newval)
m     -(int) setHighestValue : (double) newval
pas   integer set_highestValue( newval: double): integer
vb    function set_highestValue( ByVal newval As Double) As Integer
cs    int set_highestValue( double newval)
dnp   int set_highestValue( double newval)
java  int set_highestValue( double newval)
uwp   async Task<int> set_highestValue( double newval)
py    set_highestValue( newval)
php   function set_highestValue( $newval)
es    async set_highestValue( newval)
cmd   YLatitude target set_highestValue newval
```

Can be used to reset the value returned by get_lowestValue().

**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.

## latitude→set_logFrequency()
## latitude→setLogFrequency()

**YLatitude**

Changes the datalogger recording frequency for this function.

```
js      function set_logFrequency( newval)
cpp     int set_logFrequency( string newval)
m       -(int) setLogFrequency : (NSString*) newval
pas     integer set_logFrequency( newval: string): integer
vb      function set_logFrequency( ByVal newval As String) As Integer
cs      int set_logFrequency( string newval)
dnp     int set_logFrequency( string newval)
java    int set_logFrequency( String newval)
uwp     async Task<int> set_logFrequency( string newval)
py      set_logFrequency( newval)
php     function set_logFrequency( $newval)
es      async set_logFrequency( newval)
cmd     YLatitude 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". Note that setting the datalogger recording frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**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.

## latitude→set_logicalName() YLatitude
## latitude→setLogicalName()

Changes the logical name of the latitude sensor.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| cpp | int **set_logicalName**( string **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | integer **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| dnp | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| uwp | async Task<int> **set_logicalName**( string **newval**) |
| py | **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| es | async **set_logicalName**( **newval**) |
| cmd | YLatitude **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 latitude sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## latitude→set_lowestValue() YLatitude
## latitude→setLowestValue()

Changes the recorded minimal value observed.

```
js      function set_lowestValue( newval)
cpp     int set_lowestValue( double newval)
m       -(int) setLowestValue : (double) newval
pas     integer set_lowestValue( newval: double): integer
vb      function set_lowestValue( ByVal newval As Double) As Integer
cs      int set_lowestValue( double newval)
dnp     int set_lowestValue( double newval)
java    int set_lowestValue( double newval)
uwp     async Task<int> set_lowestValue( double newval)
py      set_lowestValue( newval)
php     function set_lowestValue( $newval)
es      async set_lowestValue( newval)
cmd     YLatitude target set_lowestValue newval
```

Can be used to reset the value returned by get_lowestValue().

**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.

## latitude→set_reportFrequency() YLatitude
## latitude→setReportFrequency()

Changes the timed value notification frequency for this function.

```
js      function set_reportFrequency( newval)
cpp     int set_reportFrequency( string newval)
m       -(int) setReportFrequency : (NSString*) newval
pas     integer set_reportFrequency( newval: string): integer
vb      function set_reportFrequency( ByVal newval As String) As Integer
cs      int set_reportFrequency( string newval)
dnp     int set_reportFrequency( string newval)
java    int set_reportFrequency( String newval)
uwp     async Task<int> set_reportFrequency( string newval)
py      set_reportFrequency( newval)
php     function set_reportFrequency( $newval)
es      async set_reportFrequency( newval)
cmd     YLatitude 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 (e.g. "4/h"). To disable timed value notifications for this function, use the value "OFF". Note that setting the timed value notification frequency to a greater value than the sensor native sampling frequency is unless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**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.

## latitude→set_resolution() YLatitude
## latitude→setResolution()

Changes the resolution of the measured physical values.

```
js      function set_resolution( newval)
cpp     int set_resolution( double newval)
m       -(int) setResolution : (double) newval
pas     integer set_resolution( newval: double): integer
vb      function set_resolution( ByVal newval As Double) As Integer
cs      int set_resolution( double newval)
dnp     int set_resolution( double newval)
java    int set_resolution( double newval)
uwp     async Task<int> set_resolution( double newval)
py      set_resolution( newval)
php     function set_resolution( $newval)
es      async set_resolution( newval)
cmd     YLatitude target set_resolution newval
```

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**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.

## latitude→set_userData() YLatitude
## latitude→setUserData()

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

| | |
|---|---|
| js | function **set_userData**( **data**) |
| cpp | void **set_userData**( void * **data**) |
| m | -(void) setUserData : (id) **data** |
| pas | **set_userData**( **data**: Tobject) |
| vb | procedure **set_userData**( ByVal **data** As Object) |
| cs | void **set_userData**( object **data**) |
| java | void **set_userData**( Object **data**) |
| py | **set_userData**( **data**) |
| php | function **set_userData**( **$data**) |
| es | async **set_userData**( **data**) |

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

**Parameters :**

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

## latitude→startDataLogger() YLatitude

Starts the data logger on the device.

```
js    function startDataLogger( )
cpp   int startDataLogger( )
m     -(int) startDataLogger
pas   LongInt startDataLogger( ): LongInt
vb    function startDataLogger( ) As Integer
cs    int startDataLogger( )
dnp   int startDataLogger( )
java  int startDataLogger( )
uwp   async Task<int> startDataLogger( )
py    startDataLogger( )
php   function startDataLogger( )
es    async startDataLogger( )
cmd   YLatitude target startDataLogger
```

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

**Returns :**
YAPI_SUCCESS if the call succeeds.

## latitude→stopDataLogger() YLatitude

Stops the datalogger on the device.

| | |
|---|---|
| js | function **stopDataLogger**( ) |
| cpp | int **stopDataLogger**( ) |
| m | -(int) **stopDataLogger** |
| pas | LongInt **stopDataLogger**( ): LongInt |
| vb | function **stopDataLogger**( ) As Integer |
| cs | int **stopDataLogger**( ) |
| dnp | int **stopDataLogger**( ) |
| java | int **stopDataLogger**( ) |
| uwp | async Task<int> **stopDataLogger**( ) |
| py | **stopDataLogger**( ) |
| php | function **stopDataLogger**( ) |
| es | async **stopDataLogger**( ) |
| cmd | YLatitude **target stopDataLogger** |

**Returns :**
YAPI_SUCCESS if the call succeeds.

## latitude→unmuteValueCallbacks() YLatitude

Re-enables the propagation of every new advertised value to the parent hub.

```
js      function unmuteValueCallbacks( )
cpp     int unmuteValueCallbacks( )
m       -(int) unmuteValueCallbacks
pas     LongInt unmuteValueCallbacks( ): LongInt
vb      function unmuteValueCallbacks( ) As Integer
cs      int unmuteValueCallbacks( )
dnp     int unmuteValueCallbacks( )
java    int unmuteValueCallbacks( )
uwp     async Task<int> unmuteValueCallbacks( )
py      unmuteValueCallbacks( )
php     function unmuteValueCallbacks( )
es      async unmuteValueCallbacks( )
cmd     YLatitude target unmuteValueCallbacks
```

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Returns :**
`YAPI_SUCCESS` when the call succeeds.

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

## latitude→wait_async() YLatitude

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

```
js  function wait_async( callback, context)
es  wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

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

**Returns :**

nothing.

# 22.5. Class YLongitude

Longitude sensor control interface, available for instance in the Yocto-GPS

The `YLongitude` class allows you to read and configure Yoctopuce longitude sensors. It inherits from `YSensor` class the core functions to read measurements, to register callback functions, and to access the autonomous datalogger.

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

| | |
|---|---|
| es | in HTML: <script src="../../lib/yocto_longitude.js"></script><br>in node.js: require('yoctolib-es2017/yocto_longitude.js'); |
| js | <script type='text/javascript' src='yocto_longitude.js'></script> |
| cpp | #include "yocto_longitude.h" |
| m | #import "yocto_longitude.h" |
| pas | uses yocto_longitude; |
| vb | yocto_longitude.vb |
| cs | yocto_longitude.cs |
| dnp | import YoctoProxyAPI.YLongitudeProxy |
| java | import com.yoctopuce.YoctoAPI.YLongitude; |
| uwp | import com.yoctopuce.YoctoAPI.YLongitude; |
| py | from yocto_longitude import * |
| php | require_once('yocto_longitude.php'); |
| vi | YLongitude.vi |

## Global functions

**YLongitude.FindLongitude**(**func**)

Retrieves a longitude sensor for a given identifier.

**YLongitude.FindLongitudeInContext**(**yctx**, **func**)

Retrieves a longitude sensor for a given identifier in a YAPI context.

**YLongitude.FirstLongitude**()

Starts the enumeration of longitude sensors currently accessible.

**YLongitude.FirstLongitudeInContext**(**yctx**)

Starts the enumeration of longitude sensors currently accessible.

**YLongitude.GetSimilarFunctions**()

Enumerates all functions of type Longitude available on the devices currently reachable by the library, and returns their unique hardware ID.

## YLongitude properties

**longitude→AdvMode** *[writable]*

Measuring mode used for the advertised value pushed to the parent hub.

**longitude→AdvertisedValue** *[read-only]*

Short string representing the current state of the function.

**longitude→FriendlyName** *[read-only]*

Global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

**longitude→FunctionId** *[read-only]*

Hardware identifier of the sensor, without reference to the module.

**longitude→HardwareId** *[read-only]*

Unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`.

**longitude→IsOnline** *[read-only]*

Checks if the function is currently reachable.

**longitude→LogFrequency** *[writable]*

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

**longitude→LogicalName** *[writable]*

Logical name of the function.

**longitude→ReportFrequency** *[writable]*

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

**longitude→Resolution** *[writable]*

Resolution of the measured values.

**longitude→SerialNumber** *[read-only]*

Serial number of the module, as set by the factory.

### YLongitude methods

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

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

**longitude→clearCache()**

Invalidates the cache.

**longitude→describe()**

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

**longitude→get_advMode()**

Returns the measuring mode used for the advertised value pushed to the parent hub.

**longitude→get_advertisedValue()**

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

**longitude→get_currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor, in deg/1000, as a floating point number.

**longitude→get_currentValue()**

Returns the current value of the longitude, in deg/1000, as a floating point number.

**longitude→get_dataLogger()**

Returns the `YDatalogger` object of the device hosting the sensor.

**longitude→get_errorMessage()**

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

**longitude→get_errorType()**

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

**longitude→get_friendlyName()**

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

**longitude→get_functionDescriptor()**

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

**longitude→get_functionId()**

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

**longitude→get_hardwareId()**

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

**longitude→get_highestValue()**

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

**longitude→get_logFrequency**()

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

**longitude→get_logicalName**()

Returns the logical name of the longitude sensor.

**longitude→get_lowestValue**()

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

**longitude→get_module**()

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

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

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

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

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

**longitude→get_reportFrequency**()

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

**longitude→get_resolution**()

Returns the resolution of the measured values.

**longitude→get_sensorState**()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

**longitude→get_serialNumber**()

Returns the serial number of the module, as set by the factory.

**longitude→get_unit**()

Returns the measuring unit for the longitude.

**longitude→get_userData**()

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

**longitude→isOnline**()

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

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

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

**longitude→isReadOnly**()

Test if the function is readOnly.

**longitude→isSensorReady**()

Checks if the sensor is currently able to provide an up-to-date measure.

**longitude→load**(**msValidity**)

Preloads the longitude sensor cache with a specified validity duration.

**longitude→loadAttribute**(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

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

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

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

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

**longitude→muteValueCallbacks**()

Disables the propagation of every new advertised value to the parent hub.

**longitude→nextLongitude**()

Continues the enumeration of longitude sensors started using `yFirstLongitude()`.

**longitude→registerTimedReportCallback**(**callback**)

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

**longitude→registerValueCallback**(**callback**)

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

**longitude→set_advMode**(**newval**)

Changes the measuring mode used for the advertised value pushed to the parent hub.

**longitude→set_highestValue**(**newval**)

Changes the recorded maximal value observed.

**longitude→set_logFrequency**(**newval**)

Changes the datalogger recording frequency for this function.

**longitude→set_logicalName**(**newval**)

Changes the logical name of the longitude sensor.

**longitude→set_lowestValue**(**newval**)

Changes the recorded minimal value observed.

**longitude→set_reportFrequency**(**newval**)

Changes the timed value notification frequency for this function.

**longitude→set_resolution**(**newval**)

Changes the resolution of the measured physical values.

**longitude→set_userData**(**data**)

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

**longitude→startDataLogger**()

Starts the data logger on the device.

**longitude→stopDataLogger**()

Stops the datalogger on the device.

**longitude→unmuteValueCallbacks**()

Re-enables the propagation of every new advertised value to the parent hub.

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

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

## YLongitude.FindLongitude() YLongitude
## YLongitude.FindLongitude()

Retrieves a longitude sensor for a given identifier.

| | |
|---|---|
| js | function **yFindLongitude**( **func**) |
| cpp | YLongitude* **yFindLongitude**( string **func**) |
| m | +(YLongitude*) **FindLongitude** : (NSString*) **func** |
| pas | TYLongitude **yFindLongitude**( **func**: string): TYLongitude |
| vb | function **yFindLongitude**( ByVal **func** As String) As YLongitude |
| cs | static YLongitude **FindLongitude**( string **func**) |
| dnp | static YLongitudeProxy **FindLongitude**( string **func**) |
| java | static YLongitude **FindLongitude**( String **func**) |
| uwp | static YLongitude **FindLongitude**( string **func**) |
| py | **FindLongitude**( **func**) |
| php | function **yFindLongitude**( **$func**) |
| es | static **FindLongitude**( **func**) |

The identifier can be specified using several formats:

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

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

If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

**Parameters :**

**func** a string that uniquely characterizes the longitude sensor, for instance `YGNSSMK1.longitude`.

**Returns :**

a `YLongitude` object allowing you to drive the longitude sensor.

## YLongitude.FindLongitudeInContext() YLongitude YLongitude.FindLongitudeInContext()

Retrieves a longitude sensor for a given identifier in a YAPI context.

```
java  static YLongitude FindLongitudeInContext( YAPIContext yctx, String func)
uwp   static YLongitude FindLongitudeInContext( YAPIContext yctx,
                                                string func)
es    static FindLongitudeInContext( yctx, func)
```

The identifier can be specified using several formats:

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

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

**Parameters :**

**yctx** a YAPI context

**func** a string that uniquely characterizes the longitude sensor, for instance `YGNSSMK1.longitude`.

**Returns :**

a `YLongitude` object allowing you to drive the longitude sensor.

## YLongitude.FirstLongitude() YLongitude
## YLongitude.FirstLongitude()

Starts the enumeration of longitude sensors currently accessible.

| | |
|---|---|
| js | function **yFirstLongitude**( ) |
| cpp | YLongitude * **yFirstLongitude**( ) |
| m | +(YLongitude*) **FirstLongitude** |
| pas | TYLongitude **yFirstLongitude**( ): TYLongitude |
| vb | function **yFirstLongitude**( ) As YLongitude |
| cs | static YLongitude **FirstLongitude**( ) |
| java | static YLongitude **FirstLongitude**( ) |
| uwp | static YLongitude **FirstLongitude**( ) |
| py | **FirstLongitude**( ) |
| php | function **yFirstLongitude**( ) |
| es | static **FirstLongitude**( ) |

Use the method `YLongitude.nextLongitude()` to iterate on next longitude sensors.

**Returns :**

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

## YLongitude.FirstLongitudeInContext() YLongitude YLongitude.FirstLongitudeInContext()

Starts the enumeration of longitude sensors currently accessible.

| | |
|---|---|
| java | static YLongitude **FirstLongitudeInContext**( YAPIContext **yctx**) |
| uwp | static YLongitude **FirstLongitudeInContext**( YAPIContext **yctx**) |
| es | static **FirstLongitudeInContext**( **yctx**) |

Use the method `YLongitude.nextLongitude()` to iterate on next longitude sensors.

**Parameters :**

**yctx** a YAPI context.

**Returns :**

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

## YLongitude.GetSimilarFunctions() YLongitude YLongitude.GetSimilarFunctions()

Enumerates all functions of type Longitude available on the devices currently reachable by the library, and returns their unique hardware ID.

dnp static new string[] **GetSimilarFunctions**( )

Each of these IDs can be provided as argument to the method `YLongitude.FindLongitude` to obtain an object that can control the corresponding device.

**Returns :**
an array of strings, each string containing the unique hardwareId of a device function currently connected.

## longitude→AdvMode YLongitude

Measuring mode used for the advertised value pushed to the parent hub.

dnp int **AdvMode**

**Possible values:**

Y_ADVMODE_INVALID = 0
Y_ADVMODE_IMMEDIATE = 1
Y_ADVMODE_PERIOD_AVG = 2
Y_ADVMODE_PERIOD_MIN = 3
Y_ADVMODE_PERIOD_MAX = 4

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

## longitude→AdvertisedValue — YLongitude

Short string representing the current state of the function.

dnp string **AdvertisedValue**

## longitude→FriendlyName — YLongitude

Global identifier of the function in the format MODULE_NAME.FUNCTION_NAME.

dnp string **FriendlyName**

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

## longitude→FunctionId — YLongitude

Hardware identifier of the sensor, without reference to the module.

dnp string **FunctionId**

For example `relay1`

## longitude→HardwareId — YLongitude

Unique hardware identifier of the function in the form SERIAL.FUNCTIONID.

dnp string **HardwareId**

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

## longitude→IsOnline YLongitude

Checks if the function is currently reachable.

dnp bool **IsOnline**

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

## longitude→LogFrequency YLongitude

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

dnp string **LogFrequency**

**Writable**. Changes the datalogger recording frequency for this function. 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". Note that setting the datalogger recording frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

## longitude→LogicalName — YLongitude

Logical name of the function.

dnp string **LogicalName**

**Writable**. 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.

## longitude→ReportFrequency YLongitude

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

dnp string **ReportFrequency**

**Writable**. Changes the timed value notification frequency for this function. The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (e.g. "4/h"). To disable timed value notifications for this function, use the value "OFF". Note that setting the timed value notification frequency to a greater value than the sensor native sampling frequency is unless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

## longitude→Resolution — YLongitude

Resolution of the measured values.

dnp double **Resolution**

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Writable**. Changes the resolution of the measured physical values. The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

## longitude→SerialNumber — YLongitude

Serial number of the module, as set by the factory.

dnp string **SerialNumber**

## longitude→calibrateFromPoints() YLongitude

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

```
js    function calibrateFromPoints( rawValues, refValues)
cpp   int calibrateFromPoints( vector<double> rawValues,
                               vector<double> refValues)
m     -(int) calibrateFromPoints : (NSMutableArray*) rawValues
                                 : (NSMutableArray*) refValues
pas   LongInt calibrateFromPoints( rawValues: TDoubleArray,
                                   refValues: TDoubleArray): LongInt
vb    procedure calibrateFromPoints( )
cs    int calibrateFromPoints( List<double> rawValues,
                               List<double> refValues)
dnp   int calibrateFromPoints( )
java  int calibrateFromPoints( ArrayList<Double> rawValues,
                               ArrayList<Double> refValues)
uwp   async Task<int> calibrateFromPoints( List<double> rawValues,
                                           List<double> refValues)
py    calibrateFromPoints( rawValues, refValues)
php   function calibrateFromPoints( $rawValues, $refValues)
es    async calibrateFromPoints( rawValues, refValues)
cmd   YLongitude 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.

## longitude→clearCache() YLongitude

Invalidates the cache.

| | |
|---|---|
| js | function **clearCache**( ) |
| cpp | void **clearCache**( ) |
| m | -(void) **clearCache** |
| pas | **clearCache**( ) |
| vb | procedure **clearCache**( ) |
| cs | void **clearCache**( ) |
| java | void **clearCache**( ) |
| py | **clearCache**( ) |
| php | function **clearCache**( ) |
| es | async **clearCache**( ) |

Invalidates the cache of the longitude sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

## longitude→describe() YLongitude

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

```
js      function describe( )
cpp     string describe( )
m       -(NSString*) describe
pas     string describe( ): string
vb      function describe( ) As String
cs      string describe( )
java    String describe( )
py      describe( )
php     function describe( )
es      async describe( )
```

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

**Returns :**
a string that describes the longitude sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

## longitude→get_advMode() YLongitude
## longitude→advMode()

Returns the measuring mode used for the advertised value pushed to the parent hub.

| | |
|---|---|
| js | function **get_advMode**( ) |
| cpp | Y_ADVMODE_enum **get_advMode**( ) |
| m | -(Y_ADVMODE_enum) advMode |
| pas | Integer **get_advMode**( ): Integer |
| vb | function **get_advMode**( ) As Integer |
| cs | int **get_advMode**( ) |
| dnp | int **get_advMode**( ) |
| java | int **get_advMode**( ) |
| uwp | async Task<int> **get_advMode**( ) |
| py | **get_advMode**( ) |
| php | function **get_advMode**( ) |
| es | async **get_advMode**( ) |
| cmd | YLongitude **target get_advMode** |

**Returns :**

a value among `Y_ADVMODE_IMMEDIATE`, `Y_ADVMODE_PERIOD_AVG`, `Y_ADVMODE_PERIOD_MIN` and `Y_ADVMODE_PERIOD_MAX` corresponding to the measuring mode used for the advertised value pushed to the parent hub

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

## longitude→get_advertisedValue() YLongitude
## longitude→advertisedValue()

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

```
js      function get_advertisedValue( )
cpp     string get_advertisedValue( )
m       -(NSString*) advertisedValue
pas     string get_advertisedValue( ): string
vb      function get_advertisedValue( ) As String
cs      string get_advertisedValue( )
dnp     string get_advertisedValue( )
java    String get_advertisedValue( )
uwp     async Task<string> get_advertisedValue( )
py      get_advertisedValue( )
php     function get_advertisedValue( )
es      async get_advertisedValue( )
cmd     YLongitude target get_advertisedValue
```

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

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

## longitude→get_currentRawValue() YLongitude
## longitude→currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in deg/1000, as a floating point number.

| | |
|---|---|
| js | function **get_currentRawValue**( ) |
| cpp | double **get_currentRawValue**( ) |
| m | -(double) currentRawValue |
| pas | double **get_currentRawValue**( ): double |
| vb | function **get_currentRawValue**( ) As Double |
| cs | double **get_currentRawValue**( ) |
| dnp | double **get_currentRawValue**( ) |
| java | double **get_currentRawValue**( ) |
| uwp | async Task<double> **get_currentRawValue**( ) |
| py | **get_currentRawValue**( ) |
| php | function **get_currentRawValue**( ) |
| es | async **get_currentRawValue**( ) |
| cmd | YLongitude **target get_currentRawValue** |

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in deg/1000, as a floating point number

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

## longitude→get_currentValue()
## longitude→currentValue()

YLongitude

Returns the current value of the longitude, in deg/1000, as a floating point number.

```
js      function get_currentValue( )
cpp     double get_currentValue( )
m       -(double) currentValue
pas     double get_currentValue( ): double
vb      function get_currentValue( ) As Double
cs      double get_currentValue( )
dnp     double get_currentValue( )
java    double get_currentValue( )
uwp     async Task<double> get_currentValue( )
py      get_currentValue( )
php     function get_currentValue( )
es      async get_currentValue( )
cmd     YLongitude target get_currentValue
```

Note that a get_currentValue() call will *not* start a measure in the device, it will just return the last measure that occurred in the device. Indeed, internally, each Yoctopuce devices is continuously making measurements at a hardware specific frequency.

If continuously calling get_currentValue() leads you to performances issues, then you might consider to switch to callback programming model. Check the "advanced programming" chapter in in your device user manual for more information.

**Returns :**

a floating point number corresponding to the current value of the longitude, in deg/1000, as a floating point number

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

## longitude→get_dataLogger() YLongitude
## longitude→dataLogger()

Returns the `YDatalogger` object of the device hosting the sensor.

| | |
|---|---|
| js | function **get_dataLogger**( ) |
| cpp | YDataLogger* **get_dataLogger**( ) |
| m | -(YDataLogger*) dataLogger |
| pas | TYDataLogger **get_dataLogger**( ): TYDataLogger |
| vb | function **get_dataLogger**( ) As YDataLogger |
| cs | YDataLogger **get_dataLogger**( ) |
| dnp | YDataLoggerProxy **get_dataLogger**( ) |
| java | YDataLogger **get_dataLogger**( ) |
| uwp | async Task<YDataLogger> **get_dataLogger**( ) |
| py | **get_dataLogger**( ) |
| php | function **get_dataLogger**( ) |
| es | async **get_dataLogger**( ) |

This method returns an object that can control global parameters of the data logger. The returned object should not be freed.

**Returns :**
an `YDatalogger` object, or null on error.

## longitude→get_errorMessage() YLongitude
## longitude→errorMessage()

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

```
js    function get_errorMessage( )
cpp   string get_errorMessage( )
m     -(NSString*) errorMessage
pas   string get_errorMessage( ): string
vb    function get_errorMessage( ) As String
cs    string get_errorMessage( )
java  String get_errorMessage( )
py    get_errorMessage( )
php   function get_errorMessage( )
es    get_errorMessage( )
```

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

**Returns :**
a string corresponding to the latest error message that occured while using the longitude sensor object

## longitude→get_errorType() YLongitude
## longitude→errorType()

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

| | |
|---|---|
| js | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| m | -(YRETCODE) errorType |
| pas | YRETCODE **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | **get_errorType**( ) |
| php | function **get_errorType**( ) |
| es | **get_errorType**( ) |

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

**Returns :**
a number corresponding to the code of the latest error that occurred while using the longitude sensor object

## longitude→get_friendlyName() YLongitude
## longitude→friendlyName()

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

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| dnp | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| es | async **get_friendlyName**( ) |

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

**Returns :**

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

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

## longitude→get_functionDescriptor() YLongitude
## longitude→functionDescriptor()

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

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | YFUN_DESCR **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| es | async **get_functionDescriptor**( ) |

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

**Returns :**
an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

## longitude→get_functionId()
## longitude→functionId()

YLongitude

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

```
js      function get_functionId( )
cpp     string get_functionId( )
m       -(NSString*) functionId
vb      function get_functionId( ) As String
cs      string get_functionId( )
dnp     string get_functionId( )
java    String get_functionId( )
py      get_functionId( )
php     function get_functionId( )
es      async get_functionId( )
```

For example `relay1`

**Returns :**
a string that identifies the longitude sensor (ex: `relay1`)

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

## longitude→get_hardwareId() YLongitude
## longitude→hardwareId()

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

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| dnp | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| es | async **get_hardwareId**( ) |

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

**Returns :**
a string that uniquely identifies the longitude sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

## longitude→get_highestValue() YLongitude
## longitude→highestValue()

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

| | |
|---|---|
| js | function **get_highestValue**( ) |
| cpp | double **get_highestValue**( ) |
| m | -(double) highestValue |
| pas | double **get_highestValue**( ): double |
| vb | function **get_highestValue**( ) As Double |
| cs | double **get_highestValue**( ) |
| dnp | double **get_highestValue**( ) |
| java | double **get_highestValue**( ) |
| uwp | async Task<double> **get_highestValue**( ) |
| py | **get_highestValue**( ) |
| php | function **get_highestValue**( ) |
| es | async **get_highestValue**( ) |
| cmd | YLongitude **target get_highestValue** |

Can be reset to an arbitrary value thanks to set_highestValue().

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

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

## longitude→get_logFrequency()
## longitude→logFrequency()

YLongitude

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

| | |
|---|---|
| js | function **get_logFrequency**( ) |
| cpp | string **get_logFrequency**( ) |
| m | -(NSString*) logFrequency |
| pas | string **get_logFrequency**( ): string |
| vb | function **get_logFrequency**( ) As String |
| cs | string **get_logFrequency**( ) |
| dnp | string **get_logFrequency**( ) |
| java | String **get_logFrequency**( ) |
| uwp | async Task<string> **get_logFrequency**( ) |
| py | **get_logFrequency**( ) |
| php | function **get_logFrequency**( ) |
| es | async **get_logFrequency**( ) |
| cmd | YLongitude **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`.

## longitude→get_logicalName()
## longitude→logicalName()

YLongitude

Returns the logical name of the longitude sensor.

```
js      function get_logicalName( )
cpp     string get_logicalName( )
m       -(NSString*) logicalName
pas     string get_logicalName( ): string
vb      function get_logicalName( ) As String
cs      string get_logicalName( )
dnp     string get_logicalName( )
java    String get_logicalName( )
uwp     async Task<string> get_logicalName( )
py      get_logicalName( )
php     function get_logicalName( )
es      async get_logicalName( )
cmd     YLongitude target get_logicalName
```

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

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

## longitude→get_lowestValue() YLongitude
## longitude→lowestValue()

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

| | |
|---|---|
| js | function **get_lowestValue**( ) |
| cpp | double **get_lowestValue**( ) |
| m | -(double) lowestValue |
| pas | double **get_lowestValue**( ): double |
| vb | function **get_lowestValue**( ) As Double |
| cs | double **get_lowestValue**( ) |
| dnp | double **get_lowestValue**( ) |
| java | double **get_lowestValue**( ) |
| uwp | async Task<double> **get_lowestValue**( ) |
| py | **get_lowestValue**( ) |
| php | function **get_lowestValue**( ) |
| es | async **get_lowestValue**( ) |
| cmd | YLongitude **target get_lowestValue** |

Can be reset to an arbitrary value thanks to set_lowestValue().

**Returns :**

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

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

## longitude→get_module()
## longitude→module()

YLongitude

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

| | |
|---|---|
| js | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | TYModule **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| dnp | YModuleProxy **get_module**( ) |
| java | YModule **get_module**( ) |
| py | **get_module**( ) |
| php | function **get_module**( ) |
| es | async **get_module**( ) |

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

**Returns :**
an instance of YModule

## longitude→get_module_async() YLongitude
## longitude→module_async()

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

```
js  function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as online.

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

**Parameters :**

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

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

**Returns :**

nothing : the result is provided to the callback.

## longitude→get_recordedData() YLongitude
## longitude→recordedData()

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

```
js      function get_recordedData( startTime, endTime)
cpp     YDataSet get_recordedData( double startTime, double endTime)
m       -(YDataSet*) recordedData : (double) startTime
                                  : (double) endTime
pas     TYDataSet get_recordedData( startTime: double, endTime: double): TYDataSet
vb      function get_recordedData( ) As YDataSet
cs      YDataSet get_recordedData( double startTime, double endTime)
dnp     YDataSetProxy get_recordedData( double startTime,
                                        double endTime)
java    YDataSet get_recordedData( double startTime, double endTime)
uwp     async Task<YDataSet> get_recordedData( double startTime,
                                               double endTime)
py      get_recordedData( startTime, endTime)
php     function get_recordedData( $startTime, $endTime)
es      async get_recordedData( startTime, endTime)
cmd     YLongitude 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 YDataSet 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 YDataSet 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 measure, 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 measure, 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.

## longitude→get_reportFrequency() YLongitude
## longitude→reportFrequency()

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

| | |
|---|---|
| js | function **get_reportFrequency**( ) |
| cpp | string **get_reportFrequency**( ) |
| m | -(NSString*) reportFrequency |
| pas | string **get_reportFrequency**( ): string |
| vb | function **get_reportFrequency**( ) As String |
| cs | string **get_reportFrequency**( ) |
| dnp | string **get_reportFrequency**( ) |
| java | String **get_reportFrequency**( ) |
| uwp | async Task<string> **get_reportFrequency**( ) |
| py | **get_reportFrequency**( ) |
| php | function **get_reportFrequency**( ) |
| es | async **get_reportFrequency**( ) |
| cmd | YLongitude **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`.

## longitude→get_resolution()
## longitude→resolution()

YLongitude

Returns the resolution of the measured values.

| | |
|---|---|
| js | function **get_resolution**( ) |
| cpp | double **get_resolution**( ) |
| m | -(double) resolution |
| pas | double **get_resolution**( ): double |
| vb | function **get_resolution**( ) As Double |
| cs | double **get_resolution**( ) |
| dnp | double **get_resolution**( ) |
| java | double **get_resolution**( ) |
| uwp | async Task<double> **get_resolution**( ) |
| py | **get_resolution**( ) |
| php | function **get_resolution**( ) |
| es | async **get_resolution**( ) |
| cmd | YLongitude **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. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

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

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

## longitude→get_sensorState()
## longitude→sensorState()

YLongitude

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

```
js      function get_sensorState( )
cpp     int get_sensorState( )
m       -(int) sensorState
pas     LongInt get_sensorState( ): LongInt
vb      function get_sensorState( ) As Integer
cs      int get_sensorState( )
dnp     int get_sensorState( )
java    int get_sensorState( )
uwp     async Task<int> get_sensorState( )
py      get_sensorState( )
php     function get_sensorState( )
es      async get_sensorState( )
cmd     YLongitude target get_sensorState
```

**Returns :**

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

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

## longitude→get_serialNumber()
## longitude→serialNumber()

YLongitude

Returns the serial number of the module, as set by the factory.

```
js      function get_serialNumber( )
cpp     string get_serialNumber( )
m       -(NSString*) serialNumber
pas     string get_serialNumber( ): string
vb      function get_serialNumber( ) As String
cs      string get_serialNumber( )
dnp     string get_serialNumber( )
java    String get_serialNumber( )
uwp     async Task<string> get_serialNumber( )
py      get_serialNumber( )
php     function get_serialNumber( )
es      async get_serialNumber( )
cmd     YLongitude 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 YModule.SERIALNUMBER_INVALID.

## longitude→get_unit()
## longitude→unit()

YLongitude

Returns the measuring unit for the longitude.

```
js      function get_unit( )
cpp     string get_unit( )
m       -(NSString*) unit
pas     string get_unit( ): string
vb      function get_unit( ) As String
cs      string get_unit( )
dnp     string get_unit( )
java    String get_unit( )
uwp     async Task<string> get_unit( )
py      get_unit( )
php     function get_unit( )
es      async get_unit( )
cmd     YLongitude target get_unit
```

**Returns :**
a string corresponding to the measuring unit for the longitude

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

## longitude→get_userData()
## longitude→userData()

YLongitude

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

```
js      function get_userData( )
cpp     void * get_userData( )
m       -(id) userData
pas     Tobject get_userData( ): Tobject
vb      function get_userData( ) As Object
cs      object get_userData( )
java    Object get_userData( )
py      get_userData( )
php     function get_userData( )
es      async get_userData( )
```

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

**Returns :**
the object stored previously by the caller.

## longitude→isOnline() YLongitude

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

| | |
|---|---|
| js | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | boolean **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| dnp | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | **isOnline**( ) |
| php | function **isOnline**( ) |
| es | async **isOnline**( ) |

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

**Returns :**
`true` if the longitude sensor can be reached, and `false` otherwise

## longitude→isOnline_async() YLongitude

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

```
js  function isOnline_async( callback, context)
```

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

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

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

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

**Returns :**

nothing : the result is provided to the callback.

## longitude→isReadOnly() YLongitude

Test if the function is readOnly.

| | |
|---|---|
| cpp | bool **isReadOnly**( ) |
| m | -(bool) **isReadOnly** |
| pas | boolean **isReadOnly**( ): boolean |
| vb | function **isReadOnly**( ) As Boolean |
| cs | bool **isReadOnly**( ) |
| dnp | bool **isReadOnly**( ) |
| java | boolean **isReadOnly**( ) |
| uwp | async Task<bool> **isReadOnly**( ) |
| py | **isReadOnly**( ) |
| php | function **isReadOnly**( ) |
| es | async **isReadOnly**( ) |
| cmd | YLongitude **target isReadOnly** |

Return `true` if the function is write protected or that the function is not available.

**Returns :**
`true` if the function is readOnly or not online.

## longitude→isSensorReady() YLongitude

Checks if the sensor is currently able to provide an up-to-date measure.

```
cmd YLongitude target isSensorReady
```

Returns false if the device is unreachable, or if the sensor does not have a current measure to transmit. No exception is raised if there is an error while trying to contact the device hosting $THEFUNCTION$.

**Returns :**
`true` if the sensor can provide an up-to-date measure, and `false` otherwise

## longitude→load() YLongitude

Preloads the longitude sensor cache with a specified validity duration.

| | |
|---|---|
| js | function **load**( **msValidity**) |
| cpp | YRETCODE **load**( int **msValidity**) |
| m | -(YRETCODE) **load** : (u64) **msValidity** |
| pas | YRETCODE **load**( **msValidity**: u64): YRETCODE |
| vb | function **load**( ByVal **msValidity** As Long) As YRETCODE |
| cs | YRETCODE **load**( ulong **msValidity**) |
| java | int **load**( long **msValidity**) |
| py | **load**( **msValidity**) |
| php | function **load**( **$msValidity**) |
| es | async **load**( **msValidity**) |

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

**Parameters :**

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

**Returns :**

YAPI_SUCCESS when the call succeeds.

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

## longitude→loadAttribute() YLongitude

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
js      function loadAttribute( attrName)
cpp     string loadAttribute( string attrName)
m       -(NSString*) loadAttribute : (NSString*) attrName
pas     string loadAttribute( attrName: string): string
vb      function loadAttribute( ) As String
cs      string loadAttribute( string attrName)
dnp     string loadAttribute( string attrName)
java    String loadAttribute( String attrName)
uwp     async Task<string> loadAttribute( string attrName)
py      loadAttribute( attrName)
php     function loadAttribute( $attrName)
es      async loadAttribute( attrName)
```

**Parameters :**

**attrName** the name of the requested attribute

**Returns :**

a string with the value of the the attribute

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

## longitude→loadCalibrationPoints() YLongitude

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

```
js    function loadCalibrationPoints( rawValues, refValues)
cpp   int loadCalibrationPoints( vector<double> rawValues,
                                 vector<double> refValues)
m     -(int) loadCalibrationPoints : (NSMutableArray*) rawValues
                                   : (NSMutableArray*) refValues
pas   LongInt loadCalibrationPoints( var rawValues: TDoubleArray,
                                     var refValues: TDoubleArray): LongInt
vb    procedure loadCalibrationPoints( )
cs    int loadCalibrationPoints( List<double> rawValues,
                                 List<double> refValues)
dnp   int loadCalibrationPoints( )
java  int loadCalibrationPoints( ArrayList<Double> rawValues,
                                 ArrayList<Double> refValues)
uwp   async Task<int> loadCalibrationPoints( List<double> rawValues,
                                             List<double> refValues)
py    loadCalibrationPoints( rawValues, refValues)
php   function loadCalibrationPoints( &$rawValues, &$refValues)
es    async loadCalibrationPoints( rawValues, refValues)
cmd   YLongitude 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.

## longitude→load_async() YLongitude

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

```
js  function load_async( msValidity, callback, context)
```

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

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking the JavaScript virtual machine.

**Parameters :**

| | |
|---|---|
| **msValidity** | an integer corresponding to the validity of the loaded function parameters, in milliseconds |
| **callback** | callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or YAPI_SUCCESS) |
| **context** | caller-specific object that is passed as-is to the callback function |

**Returns :**

nothing : the result is provided to the callback.

## longitude→muteValueCallbacks() YLongitude

Disables the propagation of every new advertised value to the parent hub.

| | |
|---|---|
| js | function **muteValueCallbacks**( ) |
| cpp | int **muteValueCallbacks**( ) |
| m | -(int) **muteValueCallbacks** |
| pas | LongInt **muteValueCallbacks**( ): LongInt |
| vb | function **muteValueCallbacks**( ) As Integer |
| cs | int **muteValueCallbacks**( ) |
| dnp | int **muteValueCallbacks**( ) |
| java | int **muteValueCallbacks**( ) |
| uwp | async Task<int> **muteValueCallbacks**( ) |
| py | **muteValueCallbacks**( ) |
| php | function **muteValueCallbacks**( ) |
| es | async **muteValueCallbacks**( ) |
| cmd | YLongitude **target muteValueCallbacks** |

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Returns :**
`YAPI_SUCCESS` when the call succeeds.

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

## longitude→nextLongitude() YLongitude

Continues the enumeration of longitude sensors started using `yFirstLongitude()`.

| | |
|---|---|
| js | function **nextLongitude**( ) |
| cpp | YLongitude * **nextLongitude**( ) |
| m | -(YLongitude*) **nextLongitude** |
| pas | TYLongitude **nextLongitude**( ): TYLongitude |
| vb | function **nextLongitude**( ) As YLongitude |
| cs | YLongitude **nextLongitude**( ) |
| java | YLongitude **nextLongitude**( ) |
| uwp | YLongitude **nextLongitude**( ) |
| py | **nextLongitude**( ) |
| php | function **nextLongitude**( ) |
| es | **nextLongitude**( ) |

Caution: You can't make any assumption about the returned longitude sensors order. If you want to find a specific a longitude sensor, use `Longitude.findLongitude()` and a hardwareID or a logical name.

**Returns :**

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

## longitude→registerTimedReportCallback() YLongitude

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

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| cpp | int **registerTimedReportCallback**( YLongitudeTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YLongitudeTimedReportCallback) **callback** |
| pas | LongInt **registerTimedReportCallback**( **callback**: TYLongitudeTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| uwp | async Task<int> **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| es | async **registerTimedReportCallback**( **callback**) |

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

**Parameters :**

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

## longitude→registerValueCallback() YLongitude

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

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| cpp | int **registerValueCallback**( YLongitudeValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YLongitudeValueCallback) **callback** |
| pas | LongInt **registerValueCallback**( **callback**: TYLongitudeValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| uwp | async Task<int> **registerValueCallback**( ValueCallback **callback**) |
| py | **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| es | async **registerValueCallback**( **callback**) |

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

**Parameters :**

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

## longitude→set_advMode() YLongitude
## longitude→setAdvMode()

Changes the measuring mode used for the advertised value pushed to the parent hub.

```
js    function set_advMode( newval)
cpp   int set_advMode( Y_ADVMODE_enum newval)
m     -(int) setAdvMode : (Y_ADVMODE_enum) newval
pas   integer set_advMode( newval: Integer): integer
vb    function set_advMode( ByVal newval As Integer) As Integer
cs    int set_advMode( int newval)
dnp   int set_advMode( int newval)
java  int set_advMode( int newval)
uwp   async Task<int> set_advMode( int newval)
py    set_advMode( newval)
php   function set_advMode( $newval)
es    async set_advMode( newval)
cmd   YLongitude target set_advMode newval
```

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

**Parameters :**

**newval** a value among `Y_ADVMODE_IMMEDIATE`, `Y_ADVMODE_PERIOD_AVG`, `Y_ADVMODE_PERIOD_MIN` and `Y_ADVMODE_PERIOD_MAX` corresponding to the measuring mode used for the advertised value pushed to the parent hub

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## longitude→set_highestValue() YLongitude
## longitude→setHighestValue()

Changes the recorded maximal value observed.

```
js      function set_highestValue( newval)
cpp     int set_highestValue( double newval)
m       -(int) setHighestValue : (double) newval
pas     integer set_highestValue( newval: double): integer
vb      function set_highestValue( ByVal newval As Double) As Integer
cs      int set_highestValue( double newval)
dnp     int set_highestValue( double newval)
java    int set_highestValue( double newval)
uwp     async Task<int> set_highestValue( double newval)
py      set_highestValue( newval)
php     function set_highestValue( $newval)
es      async set_highestValue( newval)
cmd     YLongitude target set_highestValue newval
```

Can be used to reset the value returned by get_lowestValue().

**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.

## longitude→set_logFrequency() YLongitude
## longitude→setLogFrequency()

Changes the datalogger recording frequency for this function.

```
js      function set_logFrequency( newval)
cpp     int set_logFrequency( string newval)
m       -(int) setLogFrequency : (NSString*) newval
pas     integer set_logFrequency( newval: string): integer
vb      function set_logFrequency( ByVal newval As String) As Integer
cs      int set_logFrequency( string newval)
dnp     int set_logFrequency( string newval)
java    int set_logFrequency( String newval)
uwp     async Task<int> set_logFrequency( string newval)
py      set_logFrequency( newval)
php     function set_logFrequency( $newval)
es      async set_logFrequency( newval)
cmd     YLongitude 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". Note that setting the datalogger recording frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**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.

## longitude→set_logicalName()
## longitude→setLogicalName()

YLongitude

Changes the logical name of the longitude sensor.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| cpp | int **set_logicalName**( string **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | integer **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| dnp | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| uwp | async Task<int> **set_logicalName**( string **newval**) |
| py | **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| es | async **set_logicalName**( **newval**) |
| cmd | YLongitude **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 longitude sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## longitude→set_lowestValue() YLongitude
## longitude→setLowestValue()

Changes the recorded minimal value observed.

```
js      function set_lowestValue( newval)
cpp     int set_lowestValue( double newval)
m       -(int) setLowestValue : (double) newval
pas     integer set_lowestValue( newval: double): integer
vb      function set_lowestValue( ByVal newval As Double) As Integer
cs      int set_lowestValue( double newval)
dnp     int set_lowestValue( double newval)
java    int set_lowestValue( double newval)
uwp     async Task<int> set_lowestValue( double newval)
py      set_lowestValue( newval)
php     function set_lowestValue( $newval)
es      async set_lowestValue( newval)
cmd     YLongitude target set_lowestValue newval
```

Can be used to reset the value returned by get_lowestValue().

**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.

## longitude→set_reportFrequency() YLongitude
## longitude→setReportFrequency()

Changes the timed value notification frequency for this function.

```
js      function set_reportFrequency( newval)
cpp     int set_reportFrequency( string newval)
m       -(int) setReportFrequency : (NSString*) newval
pas     integer set_reportFrequency( newval: string): integer
vb      function set_reportFrequency( ByVal newval As String) As Integer
cs      int set_reportFrequency( string newval)
dnp     int set_reportFrequency( string newval)
java    int set_reportFrequency( String newval)
uwp     async Task<int> set_reportFrequency( string newval)
py      set_reportFrequency( newval)
php     function set_reportFrequency( $newval)
es      async set_reportFrequency( newval)
cmd     YLongitude 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 (e.g. "4/h"). To disable timed value notifications for this function, use the value "OFF". Note that setting the timed value notification frequency to a greater value than the sensor native sampling frequency is unless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**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.

## longitude→set_resolution() YLongitude
## longitude→setResolution()

Changes the resolution of the measured physical values.

| | |
|---|---|
| js | function **set_resolution**( **newval**) |
| cpp | int **set_resolution**( double **newval**) |
| m | -(int) setResolution : (double) **newval** |
| pas | integer **set_resolution**( **newval**: double): integer |
| vb | function **set_resolution**( ByVal **newval** As Double) As Integer |
| cs | int **set_resolution**( double **newval**) |
| dnp | int **set_resolution**( double **newval**) |
| java | int **set_resolution**( double **newval**) |
| uwp | async Task<int> **set_resolution**( double **newval**) |
| py | **set_resolution**( **newval**) |
| php | function **set_resolution**( **$newval**) |
| es | async **set_resolution**( **newval**) |
| cmd | YLongitude **target set_resolution newval** |

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**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.

## longitude→set_userData()
## longitude→setUserData()

YLongitude

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

```
js      function set_userData( data)
cpp     void set_userData( void * data)
m       -(void) setUserData : (id) data
pas     set_userData( data: Tobject)
vb      procedure set_userData( ByVal data As Object)
cs      void set_userData( object data)
java    void set_userData( Object data)
py      set_userData( data)
php     function set_userData( $data)
es      async set_userData( data)
```

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

**Parameters :**

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

## longitude→startDataLogger() YLongitude

Starts the data logger on the device.

| | |
|---|---|
| js | function **startDataLogger**( ) |
| cpp | int **startDataLogger**( ) |
| m | -(int) **startDataLogger** |
| pas | LongInt **startDataLogger**( ): LongInt |
| vb | function **startDataLogger**( ) As Integer |
| cs | int **startDataLogger**( ) |
| dnp | int **startDataLogger**( ) |
| java | int **startDataLogger**( ) |
| uwp | async Task<int> **startDataLogger**( ) |
| py | **startDataLogger**( ) |
| php | function **startDataLogger**( ) |
| es | async **startDataLogger**( ) |
| cmd | YLongitude **target startDataLogger** |

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

**Returns :**
YAPI_SUCCESS if the call succeeds.

## longitude→stopDataLogger() YLongitude

Stops the datalogger on the device.

```
js      function stopDataLogger( )
cpp     int stopDataLogger( )
m       -(int) stopDataLogger
pas     LongInt stopDataLogger( ): LongInt
vb      function stopDataLogger( ) As Integer
cs      int stopDataLogger( )
dnp     int stopDataLogger( )
java    int stopDataLogger( )
uwp     async Task<int> stopDataLogger( )
py      stopDataLogger( )
php     function stopDataLogger( )
es      async stopDataLogger( )
cmd     YLongitude target stopDataLogger
```

**Returns :**
YAPI_SUCCESS if the call succeeds.

## longitude→unmuteValueCallbacks() YLongitude

Re-enables the propagation of every new advertised value to the parent hub.

| | |
|---|---|
| js | function **unmuteValueCallbacks**( ) |
| cpp | int **unmuteValueCallbacks**( ) |
| m | -(int) **unmuteValueCallbacks** |
| pas | LongInt **unmuteValueCallbacks**( ): LongInt |
| vb | function **unmuteValueCallbacks**( ) As Integer |
| cs | int **unmuteValueCallbacks**( ) |
| dnp | int **unmuteValueCallbacks**( ) |
| java | int **unmuteValueCallbacks**( ) |
| uwp | async Task<int> **unmuteValueCallbacks**( ) |
| py | **unmuteValueCallbacks**( ) |
| php | function **unmuteValueCallbacks**( ) |
| es | async **unmuteValueCallbacks**( ) |
| cmd | YLongitude **target unmuteValueCallbacks** |

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Returns :**
`YAPI_SUCCESS` when the call succeeds.

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

## longitude→wait_async() YLongitude

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

```
js  function wait_async( callback, context)
es  wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

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

**Returns :**

nothing.

# 22.6. Class YAltitude

Altimeter control interface, available for instance in the Yocto-Altimeter-V2 or the Yocto-GPS

The `YAltitude` class allows you to read and configure Yoctopuce altimeters. It inherits from `YSensor` class the core functions to read measurements, to register callback functions, and to access the autonomous datalogger. This class adds the ability to configure the barometric pressure adjusted to sea level (QNH) for barometric sensors.

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

| | |
|---|---|
| es | in HTML: <script src="../../lib/yocto_altitude.js"></script><br>in node.js: require('yoctolib-es2017/yocto_altitude.js'); |
| js | <script type='text/javascript' src='yocto_altitude.js'></script> |
| cpp | #include "yocto_altitude.h" |
| m | #import "yocto_altitude.h" |
| pas | uses yocto_altitude; |
| vb | yocto_altitude.vb |
| cs | yocto_altitude.cs |
| dnp | import YoctoProxyAPI.YAltitudeProxy |
| java | import com.yoctopuce.YoctoAPI.YAltitude; |
| uwp | import com.yoctopuce.YoctoAPI.YAltitude; |
| py | from yocto_altitude import * |
| php | require_once('yocto_altitude.php'); |
| vi | YAltitude.vi |

## Global functions

**YAltitude.FindAltitude**(**func**)

Retrieves an altimeter for a given identifier.

**YAltitude.FindAltitudeInContext**(**yctx**, **func**)

Retrieves an altimeter for a given identifier in a YAPI context.

**YAltitude.FirstAltitude**()

Starts the enumeration of altimeters currently accessible.

**YAltitude.FirstAltitudeInContext**(**yctx**)

Starts the enumeration of altimeters currently accessible.

**YAltitude.GetSimilarFunctions**()

Enumerates all functions of type Altitude available on the devices currently reachable by the library, and returns their unique hardware ID.

## YAltitude properties

**altitude→AdvMode** *[writable]*

Measuring mode used for the advertised value pushed to the parent hub.

**altitude→AdvertisedValue** *[read-only]*

Short string representing the current state of the function.

**altitude→FriendlyName** *[read-only]*

Global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

**altitude→FunctionId** *[read-only]*

Hardware identifier of the sensor, without reference to the module.

**altitude→HardwareId** *[read-only]*

Unique hardware identifier of the function in the form SERIAL.FUNCTIONID.

**altitude→IsOnline** *[read-only]*

Checks if the function is currently reachable.

**altitude→LogFrequency** *[writable]*

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

**altitude→LogicalName** *[writable]*

Logical name of the function.

**altitude→Qnh** *[writable]*

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

**altitude→ReportFrequency** *[writable]*

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

**altitude→Resolution** *[writable]*

Resolution of the measured values.

**altitude→SerialNumber** *[read-only]*

Serial number of the module, as set by the factory.

## 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→clearCache()**

Invalidates the cache.

**altitude→describe()**

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

**altitude→get_advMode()**

Returns the measuring mode used for the advertised value pushed to the parent hub.

**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_dataLogger()**

Returns the YDatalogger object of the device hosting the sensor.

**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 YDataSet 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_sensorState**()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

**altitude→get_serialNumber**()

Returns the serial number of the module, as set by the factory.

**altitude→get_technology**()

Returns the technology used by the sesnor to compute altitude.

**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→isReadOnly**()

Test if the function is readOnly.

**altitude→isSensorReady**()

Checks if the sensor is currently able to provide an up-to-date measure.

**altitude→load**(**msValidity**)

Preloads the altimeter cache with a specified validity duration.

**altitude→loadAttribute**(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

**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→muteValueCallbacks()**

Disables the propagation of every new advertised value to the parent hub.

**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_advMode(newval)**

Changes the measuring mode used for the advertised value pushed to the parent hub.

**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→startDataLogger()**

Starts the data logger on the device.

**altitude→stopDataLogger()**

Stops the datalogger on the device.

**altitude→unmuteValueCallbacks()**

Re-enables the propagation of every new advertised value to the parent hub.

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

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

## YAltitude.FindAltitude() YAltitude
## YAltitude.FindAltitude()

Retrieves an altimeter for a given identifier.

```
js      function yFindAltitude( func)
cpp     YAltitude* yFindAltitude( string func)
m       +(YAltitude*) FindAltitude : (NSString*) func
pas     TYAltitude yFindAltitude( func: string): TYAltitude
vb      function yFindAltitude( ByVal func As String) As YAltitude
cs      static YAltitude FindAltitude( string func)
dnp     static YAltitudeProxy FindAltitude( string func)
java    static YAltitude FindAltitude( String func)
uwp     static YAltitude FindAltitude( string func)
py      FindAltitude( func)
php     function yFindAltitude( $func)
es      static FindAltitude( func)
```

The identifier can be specified using several formats:

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

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

If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

**Parameters :**

**func** a string that uniquely characterizes the altimeter, for instance `YALTIMK2.altitude`.

**Returns :**

a `YAltitude` object allowing you to drive the altimeter.

## YAltitude.FindAltitudeInContext() YAltitude YAltitude.FindAltitudeInContext()

Retrieves an altimeter for a given identifier in a YAPI context.

java static YAltitude **FindAltitudeInContext**( YAPIContext **yctx**, String **func**)
uwp static YAltitude **FindAltitudeInContext**( YAPIContext **yctx**, string **func**)
es static **FindAltitudeInContext**( **yctx**, **func**)

The identifier can be specified using several formats:

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

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

**Parameters :**

**yctx** a YAPI context

**func** a string that uniquely characterizes the altimeter, for instance `YALTIMK2.altitude`.

**Returns :**

a `YAltitude` object allowing you to drive the altimeter.

## YAltitude.FirstAltitude() YAltitude
## YAltitude.FirstAltitude()

Starts the enumeration of altimeters currently accessible.

| | |
|---|---|
| js | function **yFirstAltitude**( ) |
| cpp | YAltitude * **yFirstAltitude**( ) |
| m | +(YAltitude*) **FirstAltitude** |
| pas | TYAltitude **yFirstAltitude**( ): TYAltitude |
| vb | function **yFirstAltitude**( ) As YAltitude |
| cs | static YAltitude **FirstAltitude**( ) |
| java | static YAltitude **FirstAltitude**( ) |
| uwp | static YAltitude **FirstAltitude**( ) |
| py | **FirstAltitude**( ) |
| php | function **yFirstAltitude**( ) |
| es | static **FirstAltitude**( ) |

Use the method `YAltitude.nextAltitude()` to iterate on next altimeters.

**Returns :**

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

## YAltitude.FirstAltitudeInContext() YAltitude YAltitude.FirstAltitudeInContext()

Starts the enumeration of altimeters currently accessible.

java static YAltitude **FirstAltitudeInContext**( YAPIContext **yctx**)
uwp static YAltitude **FirstAltitudeInContext**( YAPIContext **yctx**)
es static **FirstAltitudeInContext**( **yctx**)

Use the method `YAltitude.nextAltitude()` to iterate on next altimeters.

**Parameters :**

**yctx** a YAPI context.

**Returns :**

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

## YAltitude.GetSimilarFunctions() YAltitude
## YAltitude.GetSimilarFunctions()

Enumerates all functions of type Altitude available on the devices currently reachable by the library, and returns their unique hardware ID.

dnp static new string[] **GetSimilarFunctions**( )

Each of these IDs can be provided as argument to the method `YAltitude.FindAltitude` to obtain an object that can control the corresponding device.

**Returns :**
an array of strings, each string containing the unique hardwareId of a device function currently connected.

## altitude→AdvMode YAltitude

Measuring mode used for the advertised value pushed to the parent hub.

dnp int **AdvMode**

**Possible values:**

Y_ADVMODE_INVALID = 0
Y_ADVMODE_IMMEDIATE = 1
Y_ADVMODE_PERIOD_AVG = 2
Y_ADVMODE_PERIOD_MIN = 3
Y_ADVMODE_PERIOD_MAX = 4

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

## altitude→AdvertisedValue — YAltitude

Short string representing the current state of the function.

dnp string **AdvertisedValue**

## altitude→FriendlyName — YAltitude

Global identifier of the function in the format MODULE_NAME.FUNCTION_NAME.

dnp string **FriendlyName**

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

## altitude→FunctionId — YAltitude

Hardware identifier of the sensor, without reference to the module.

dnp string **FunctionId**

For example `relay1`

## altitude→HardwareId — YAltitude

Unique hardware identifier of the function in the form SERIAL.FUNCTIONID.

dnp string **HardwareId**

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

## altitude→IsOnline — YAltitude

Checks if the function is currently reachable.

dnp bool **IsOnline**

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

## altitude→LogFrequency — YAltitude

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

dnp string **LogFrequency**

**Writable**. Changes the datalogger recording frequency for this function. 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". Note that setting the datalogger recording frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

## altitude→LogicalName — YAltitude

Logical name of the function.

dnp string **LogicalName**

**Writable**. 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.

## altitude→Qnh — YAltitude

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

dnp double **Qnh**

Applicable to barometric altimeters only.

**Writable**. This enables you to compensate for atmospheric pressure changes due to weather conditions. Applicable to barometric altimeters only. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

## altitude→ReportFrequency — YAltitude

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

dnp string **ReportFrequency**

**Writable**. Changes the timed value notification frequency for this function. The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (e.g. "4/h"). To disable timed value notifications for this function, use the value "OFF". Note that setting the timed value notification frequency to a greater value than the sensor native sampling frequency is unless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

## altitude→Resolution — YAltitude

Resolution of the measured values.

dnp double **Resolution**

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Writable**. Changes the resolution of the measured physical values. The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

## altitude→SerialNumber — YAltitude

Serial number of the module, as set by the factory.

dnp string **SerialNumber**

## altitude→calibrateFromPoints() YAltitude

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

```
js    function calibrateFromPoints( rawValues, refValues)
cpp   int calibrateFromPoints( vector<double> rawValues,
                               vector<double> refValues)
m     -(int) calibrateFromPoints : (NSMutableArray*) rawValues
                                 : (NSMutableArray*) refValues
pas   LongInt calibrateFromPoints( rawValues: TDoubleArray,
                                   refValues: TDoubleArray): LongInt
vb    procedure calibrateFromPoints( )
cs    int calibrateFromPoints( List<double> rawValues,
                               List<double> refValues)
dnp   int calibrateFromPoints( )
java  int calibrateFromPoints( ArrayList<Double> rawValues,
                               ArrayList<Double> refValues)
uwp   async Task<int> calibrateFromPoints( List<double> rawValues,
                                           List<double> refValues)
py    calibrateFromPoints( rawValues, refValues)
php   function calibrateFromPoints( $rawValues, $refValues)
es    async calibrateFromPoints( rawValues, refValues)
cmd   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→clearCache() YAltitude

Invalidates the cache.

| | |
|---|---|
| js | function **clearCache**( ) |
| cpp | void **clearCache**( ) |
| m | -(void) **clearCache** |
| pas | **clearCache**( ) |
| vb | procedure **clearCache**( ) |
| cs | void **clearCache**( ) |
| java | void **clearCache**( ) |
| py | **clearCache**( ) |
| php | function **clearCache**( ) |
| es | async **clearCache**( ) |

Invalidates the cache of the altimeter attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

## altitude→describe() YAltitude

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

```
js    function describe( )
cpp   string describe( )
m     -(NSString*) describe
pas   string describe( ): string
vb    function describe( ) As String
cs    string describe( )
java  String describe( )
py    describe( )
php   function describe( )
es    async describe( )
```

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

**Returns :**
a string that describes the altimeter (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

## altitude→get_advMode()
## altitude→advMode()

YAltitude

Returns the measuring mode used for the advertised value pushed to the parent hub.

| | |
|---|---|
| js | function **get_advMode**( ) |
| cpp | Y_ADVMODE_enum **get_advMode**( ) |
| m | -(Y_ADVMODE_enum) advMode |
| pas | Integer **get_advMode**( ): Integer |
| vb | function **get_advMode**( ) As Integer |
| cs | int **get_advMode**( ) |
| dnp | int **get_advMode**( ) |
| java | int **get_advMode**( ) |
| uwp | async Task<int> **get_advMode**( ) |
| py | **get_advMode**( ) |
| php | function **get_advMode**( ) |
| es | async **get_advMode**( ) |
| cmd | YAltitude **target get_advMode** |

**Returns :**

a value among `Y_ADVMODE_IMMEDIATE`, `Y_ADVMODE_PERIOD_AVG`, `Y_ADVMODE_PERIOD_MIN` and `Y_ADVMODE_PERIOD_MAX` corresponding to the measuring mode used for the advertised value pushed to the parent hub

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

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

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

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | string **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| dnp | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| uwp | async Task<string> **get_advertisedValue**( ) |
| py | **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| es | async **get_advertisedValue**( ) |
| cmd | 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()

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

| | |
|---|---|
| js | function **get_currentRawValue**( ) |
| cpp | double **get_currentRawValue**( ) |
| m | -(double) currentRawValue |
| pas | double **get_currentRawValue**( ): double |
| vb | function **get_currentRawValue**( ) As Double |
| cs | double **get_currentRawValue**( ) |
| dnp | double **get_currentRawValue**( ) |
| java | double **get_currentRawValue**( ) |
| uwp | async Task<double> **get_currentRawValue**( ) |
| py | **get_currentRawValue**( ) |
| php | function **get_currentRawValue**( ) |
| es | async **get_currentRawValue**( ) |
| cmd | 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()
## altitude→currentValue()

YAltitude

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

```
js      function get_currentValue( )
cpp     double get_currentValue( )
m       -(double) currentValue
pas     double get_currentValue( ): double
vb      function get_currentValue( ) As Double
cs      double get_currentValue( )
dnp     double get_currentValue( )
java    double get_currentValue( )
uwp     async Task<double> get_currentValue( )
py      get_currentValue( )
php     function get_currentValue( )
es      async get_currentValue( )
cmd     YAltitude target get_currentValue
```

Note that a get_currentValue() call will *not* start a measure in the device, it will just return the last measure that occurred in the device. Indeed, internally, each Yoctopuce devices is continuously making measurements at a hardware specific frequency.

If continuously calling get_currentValue() leads you to performances issues, then you might consider to switch to callback programming model. Check the "advanced programming" chapter in in your device user manual for more information.

**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_dataLogger()
## altitude→dataLogger()

**YAltitude**

Returns the `YDatalogger` object of the device hosting the sensor.

| | |
|---|---|
| js | function **get_dataLogger**( ) |
| cpp | YDataLogger* **get_dataLogger**( ) |
| m | -(YDataLogger*) dataLogger |
| pas | TYDataLogger **get_dataLogger**( ): TYDataLogger |
| vb | function **get_dataLogger**( ) As YDataLogger |
| cs | YDataLogger **get_dataLogger**( ) |
| dnp | YDataLoggerProxy **get_dataLogger**( ) |
| java | YDataLogger **get_dataLogger**( ) |
| uwp | async Task<YDataLogger> **get_dataLogger**( ) |
| py | **get_dataLogger**( ) |
| php | function **get_dataLogger**( ) |
| es | async **get_dataLogger**( ) |

This method returns an object that can control global parameters of the data logger. The returned object should not be freed.

**Returns :**
an `YDatalogger` object, or null on error.

## altitude→get_errorMessage()
## altitude→errorMessage()

YAltitude

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

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | string **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| es | **get_errorMessage**( ) |

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

**Returns :**
a string corresponding to the latest error message that occured while using the altimeter object

## altitude→get_errorType()
## altitude→errorType()

**YAltitude**

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

| | |
|---|---|
| js | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| m | -(YRETCODE) errorType |
| pas | YRETCODE **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | **get_errorType**( ) |
| php | function **get_errorType**( ) |
| es | **get_errorType**( ) |

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

**Returns :**
a number corresponding to the code of the latest error that occurred while using the altimeter object

## altitude→get_friendlyName()
## altitude→friendlyName()

YAltitude

Returns a global identifier of the altimeter in the format MODULE_NAME.FUNCTION_NAME.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| dnp | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| es | async **get_friendlyName**( ) |

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

**Returns :**

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

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

## altitude→get_functionDescriptor() YAltitude
## altitude→functionDescriptor()

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

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | YFUN_DESCR **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| es | async **get_functionDescriptor**( ) |

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

**Returns :**
an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

## altitude→get_functionId()
## altitude→functionId()

YAltitude

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

```
js      function get_functionId( )
cpp     string get_functionId( )
m       -(NSString*) functionId
vb      function get_functionId( ) As String
cs      string get_functionId( )
dnp     string get_functionId( )
java    String get_functionId( )
py      get_functionId( )
php     function get_functionId( )
es      async get_functionId( )
```

For example `relay1`

**Returns :**
a string that identifies the altimeter (ex: `relay1`)

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

## altitude→get_hardwareId() YAltitude
## altitude→hardwareId()

Returns the unique hardware identifier of the altimeter in the form SERIAL.FUNCTIONID.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| dnp | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| es | async **get_hardwareId**( ) |

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

**Returns :**
a string that uniquely identifies the altimeter (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

## altitude→get_highestValue()
## altitude→highestValue()

YAltitude

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

| | |
|---|---|
| js | function **get_highestValue**( ) |
| cpp | double **get_highestValue**( ) |
| m | -(double) highestValue |
| pas | double **get_highestValue**( ): double |
| vb | function **get_highestValue**( ) As Double |
| cs | double **get_highestValue**( ) |
| dnp | double **get_highestValue**( ) |
| java | double **get_highestValue**( ) |
| uwp | async Task<double> **get_highestValue**( ) |
| py | **get_highestValue**( ) |
| php | function **get_highestValue**( ) |
| es | async **get_highestValue**( ) |
| cmd | YAltitude **target get_highestValue** |

Can be reset to an arbitrary value thanks to set_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()

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

| | |
|---|---|
| js | function **get_logFrequency**( ) |
| cpp | string **get_logFrequency**( ) |
| m | -(NSString*) logFrequency |
| pas | string **get_logFrequency**( ): string |
| vb | function **get_logFrequency**( ) As String |
| cs | string **get_logFrequency**( ) |
| dnp | string **get_logFrequency**( ) |
| java | String **get_logFrequency**( ) |
| uwp | async Task<string> **get_logFrequency**( ) |
| py | **get_logFrequency**( ) |
| php | function **get_logFrequency**( ) |
| es | async **get_logFrequency**( ) |
| cmd | 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()
## altitude→logicalName()

YAltitude

Returns the logical name of the altimeter.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | string **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| dnp | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| uwp | async Task<string> **get_logicalName**( ) |
| py | **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| es | async **get_logicalName**( ) |
| cmd | 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()

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

| | |
|---|---|
| js | function **get_lowestValue**( ) |
| cpp | double **get_lowestValue**( ) |
| m | -(double) lowestValue |
| pas | double **get_lowestValue**( ): double |
| vb | function **get_lowestValue**( ) As Double |
| cs | double **get_lowestValue**( ) |
| dnp | double **get_lowestValue**( ) |
| java | double **get_lowestValue**( ) |
| uwp | async Task<double> **get_lowestValue**( ) |
| py | **get_lowestValue**( ) |
| php | function **get_lowestValue**( ) |
| es | async **get_lowestValue**( ) |
| cmd | YAltitude **target get_lowestValue** |

Can be reset to an arbitrary value thanks to set_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_module()
## altitude→module()

YAltitude

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

| | |
|---|---|
| js | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | TYModule **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| dnp | YModuleProxy **get_module**( ) |
| java | YModule **get_module**( ) |
| py | **get_module**( ) |
| php | function **get_module**( ) |
| es | async **get_module**( ) |

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

**Returns :**
an instance of YModule

## altitude→get_module_async() YAltitude
## altitude→module_async()

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

```
js  function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as on-line.

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

**Parameters :**

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

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

**Returns :**

nothing : the result is provided to the callback.

## altitude→get_qnh()
## altitude→qnh()

YAltitude

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

| | |
|---|---|
| js | function **get_qnh**( ) |
| cpp | double **get_qnh**( ) |
| m | -(double) qnh |
| pas | double **get_qnh**( ): double |
| vb | function **get_qnh**( ) As Double |
| cs | double **get_qnh**( ) |
| dnp | double **get_qnh**( ) |
| java | double **get_qnh**( ) |
| uwp | async Task<double> **get_qnh**( ) |
| py | **get_qnh**( ) |
| php | function **get_qnh**( ) |
| es | async **get_qnh**( ) |
| cmd | YAltitude **target** **get_qnh** |

Applicable to barometric altimeters only.

**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()

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

```
js      function get_recordedData( startTime, endTime)
cpp     YDataSet get_recordedData( double startTime, double endTime)
m       -(YDataSet*) recordedData : (double) startTime
                                  : (double) endTime
pas     TYDataSet get_recordedData( startTime: double, endTime: double): TYDataSet
vb      function get_recordedData( ) As YDataSet
cs      YDataSet get_recordedData( double startTime, double endTime)
dnp     YDataSetProxy get_recordedData( double startTime,
                                        double endTime)
java    YDataSet get_recordedData( double startTime, double endTime)
uwp     async Task<YDataSet> get_recordedData( double startTime,
                                               double endTime)
py      get_recordedData( startTime, endTime)
php     function get_recordedData( $startTime, $endTime)
es      async get_recordedData( startTime, endTime)
cmd     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 YDataSet 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 YDataSet 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 measure, 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 measure, 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()
## altitude→reportFrequency()

YAltitude

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

```
js      function get_reportFrequency( )
cpp     string get_reportFrequency( )
m       -(NSString*) reportFrequency
pas     string get_reportFrequency( ): string
vb      function get_reportFrequency( ) As String
cs      string get_reportFrequency( )
dnp     string get_reportFrequency( )
java    String get_reportFrequency( )
uwp     async Task<string> get_reportFrequency( )
py      get_reportFrequency( )
php     function get_reportFrequency( )
es      async get_reportFrequency( )
cmd     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()
## altitude→resolution()

YAltitude

Returns the resolution of the measured values.

| | |
|---|---|
| js | function **get_resolution**( ) |
| cpp | double **get_resolution**( ) |
| m | -(double) resolution |
| pas | double **get_resolution**( ): double |
| vb | function **get_resolution**( ) As Double |
| cs | double **get_resolution**( ) |
| dnp | double **get_resolution**( ) |
| java | double **get_resolution**( ) |
| uwp | async Task<double> **get_resolution**( ) |
| py | **get_resolution**( ) |
| php | function **get_resolution**( ) |
| es | async **get_resolution**( ) |
| cmd | 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. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**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_sensorState()
## altitude→sensorState()

YAltitude

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

```
js      function get_sensorState( )
cpp     int get_sensorState( )
m       -(int) sensorState
pas     LongInt get_sensorState( ): LongInt
vb      function get_sensorState( ) As Integer
cs      int get_sensorState( )
dnp     int get_sensorState( )
java    int get_sensorState( )
uwp     async Task<int> get_sensorState( )
py      get_sensorState( )
php     function get_sensorState( )
es      async get_sensorState( )
cmd     YAltitude target get_sensorState
```

**Returns :**

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns Y_SENSORSTATE_INVALID.

## altitude→get_serialNumber() YAltitude
## altitude→serialNumber()

Returns the serial number of the module, as set by the factory.

```
js      function get_serialNumber( )
cpp     string get_serialNumber( )
m       -(NSString*) serialNumber
pas     string get_serialNumber( ): string
vb      function get_serialNumber( ) As String
cs      string get_serialNumber( )
dnp     string get_serialNumber( )
java    String get_serialNumber( )
uwp     async Task<string> get_serialNumber( )
py      get_serialNumber( )
php     function get_serialNumber( )
es      async get_serialNumber( )
cmd     YAltitude 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 YModule.SERIALNUMBER_INVALID.

## altitude→get_technology()
## altitude→technology()

YAltitude

Returns the technology used by the sesnor to compute altitude.

```
js      function get_technology( )
cpp     string get_technology( )
m       -(NSString*) technology
pas     string get_technology( ): string
vb      function get_technology( ) As String
cs      string get_technology( )
dnp     string get_technology( )
java    String get_technology( )
uwp     async Task<string> get_technology( )
py      get_technology( )
php     function get_technology( )
es      async get_technology( )
cmd     YAltitude target get_technology
```

Possibles values are "barometric" and "gps"

**Returns :**
a string corresponding to the technology used by the sesnor to compute altitude

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

## altitude→get_unit()
## altitude→unit()

YAltitude

Returns the measuring unit for the altitude.

| | |
|---|---|
| js | function **get_unit**( ) |
| cpp | string **get_unit**( ) |
| m | -(NSString*) unit |
| pas | string **get_unit**( ): string |
| vb | function **get_unit**( ) As String |
| cs | string **get_unit**( ) |
| dnp | string **get_unit**( ) |
| java | String **get_unit**( ) |
| uwp | async Task<string> **get_unit**( ) |
| py | **get_unit**( ) |
| php | function **get_unit**( ) |
| es | async **get_unit**( ) |
| cmd | 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→get_userData()
## altitude→userData()

YAltitude

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

| | |
|---|---|
| js | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(id) userData |
| pas | Tobject **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | **get_userData**( ) |
| php | function **get_userData**( ) |
| es | async **get_userData**( ) |

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

**Returns :**
the object stored previously by the caller.

## altitude→isOnline() YAltitude

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

| | |
|---|---|
| js | function **isOnline**( ) |
| cpp | bool **isOnline**( ) |
| m | -(BOOL) **isOnline** |
| pas | boolean **isOnline**( ): boolean |
| vb | function **isOnline**( ) As Boolean |
| cs | bool **isOnline**( ) |
| dnp | bool **isOnline**( ) |
| java | boolean **isOnline**( ) |
| py | **isOnline**( ) |
| php | function **isOnline**( ) |
| es | async **isOnline**( ) |

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

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

## altitude→isOnline_async() YAltitude

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

```
js  function isOnline_async( callback, context)
```

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

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

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

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

**Returns :**

nothing : the result is provided to the callback.

## altitude→isReadOnly() YAltitude

Test if the function is readOnly.

| | |
|---|---|
| cpp | bool **isReadOnly**( ) |
| m | -(bool) **isReadOnly** |
| pas | boolean **isReadOnly**( ): boolean |
| vb | function **isReadOnly**( ) As Boolean |
| cs | bool **isReadOnly**( ) |
| dnp | bool **isReadOnly**( ) |
| java | boolean **isReadOnly**( ) |
| uwp | async Task<bool> **isReadOnly**( ) |
| py | **isReadOnly**( ) |
| php | function **isReadOnly**( ) |
| es | async **isReadOnly**( ) |
| cmd | YAltitude **target isReadOnly** |

Return `true` if the function is write protected or that the function is not available.

**Returns :**
`true` if the function is readOnly or not online.

## altitude→isSensorReady() YAltitude

Checks if the sensor is currently able to provide an up-to-date measure.

cmd YAltitude **target isSensorReady**

Returns false if the device is unreachable, or if the sensor does not have a current measure to transmit. No exception is raised if there is an error while trying to contact the device hosting $THEFUNCTION$.

**Returns :**
`true` if the sensor can provide an up-to-date measure, and `false` otherwise

## altitude→load() YAltitude

Preloads the altimeter cache with a specified validity duration.

```
js      function load( msValidity)
cpp     YRETCODE load( int msValidity)
m       -(YRETCODE) load : (u64) msValidity
pas     YRETCODE load( msValidity: u64): YRETCODE
vb      function load( ByVal msValidity As Long) As YRETCODE
cs      YRETCODE load( ulong msValidity)
java    int load( long msValidity)
py      load( msValidity)
php     function load( $msValidity)
es      async load( msValidity)
```

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

**Parameters :**

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

**Returns :**

YAPI_SUCCESS when the call succeeds.

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

## altitude→loadAttribute() YAltitude

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
js      function loadAttribute( attrName)
cpp     string loadAttribute( string attrName)
m       -(NSString*) loadAttribute : (NSString*) attrName
pas     string loadAttribute( attrName: string): string
vb      function loadAttribute( ) As String
cs      string loadAttribute( string attrName)
dnp     string loadAttribute( string attrName)
java    String loadAttribute( String attrName)
uwp     async Task<string> loadAttribute( string attrName)
py      loadAttribute( attrName)
php     function loadAttribute( $attrName)
es      async loadAttribute( attrName)
```

**Parameters :**

**attrName** the name of the requested attribute

**Returns :**

a string with the value of the the attribute

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

## altitude→loadCalibrationPoints() YAltitude

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

```
js      function loadCalibrationPoints( rawValues, refValues)
cpp     int loadCalibrationPoints( vector<double> rawValues,
                                   vector<double> refValues)
m       -(int) loadCalibrationPoints : (NSMutableArray*) rawValues
                                     : (NSMutableArray*) refValues
pas     LongInt loadCalibrationPoints( var rawValues: TDoubleArray,
                                       var refValues: TDoubleArray): LongInt
vb      procedure loadCalibrationPoints( )
cs      int loadCalibrationPoints( List<double> rawValues,
                                   List<double> refValues)
dnp     int loadCalibrationPoints( )
java    int loadCalibrationPoints( ArrayList<Double> rawValues,
                                   ArrayList<Double> refValues)
uwp     async Task<int> loadCalibrationPoints( List<double> rawValues,
                                               List<double> refValues)
py      loadCalibrationPoints( rawValues, refValues)
php     function loadCalibrationPoints( &$rawValues, &$refValues)
es      async loadCalibrationPoints( rawValues, refValues)
cmd     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→load_async() YAltitude

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

```
js  function load_async( msValidity, callback, context)
```

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

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking the JavaScript virtual machine.

**Parameters :**

| | |
|---|---|
| **msValidity** | an integer corresponding to the validity of the loaded function parameters, in milliseconds |
| **callback** | callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or YAPI_SUCCESS) |
| **context** | caller-specific object that is passed as-is to the callback function |

**Returns :**

nothing : the result is provided to the callback.

## altitude→muteValueCallbacks() YAltitude

Disables the propagation of every new advertised value to the parent hub.

| | |
|---|---|
| js | function **muteValueCallbacks**( ) |
| cpp | int **muteValueCallbacks**( ) |
| m | -(int) **muteValueCallbacks** |
| pas | LongInt **muteValueCallbacks**( ): LongInt |
| vb | function **muteValueCallbacks**( ) As Integer |
| cs | int **muteValueCallbacks**( ) |
| dnp | int **muteValueCallbacks**( ) |
| java | int **muteValueCallbacks**( ) |
| uwp | async Task<int> **muteValueCallbacks**( ) |
| py | **muteValueCallbacks**( ) |
| php | function **muteValueCallbacks**( ) |
| es | async **muteValueCallbacks**( ) |
| cmd | YAltitude **target muteValueCallbacks** |

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Returns :**
`YAPI_SUCCESS` when the call succeeds.

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

## altitude→nextAltitude() YAltitude

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

| | |
|---|---|
| `js` | function **nextAltitude**( ) |
| `cpp` | YAltitude * **nextAltitude**( ) |
| `m` | -(YAltitude*) **nextAltitude** |
| `pas` | TYAltitude **nextAltitude**( ): TYAltitude |
| `vb` | function **nextAltitude**( ) As YAltitude |
| `cs` | YAltitude **nextAltitude**( ) |
| `java` | YAltitude **nextAltitude**( ) |
| `uwp` | YAltitude **nextAltitude**( ) |
| `py` | **nextAltitude**( ) |
| `php` | function **nextAltitude**( ) |
| `es` | **nextAltitude**( ) |

Caution: You can't make any assumption about the returned altimeters order. If you want to find a specific an altimeter, use `Altitude.findAltitude()` and a hardwareID or a logical name.

**Returns :**

a pointer to a `YAltitude` object, corresponding to an altimeter currently online, or a `null` pointer if there are no more altimeters to enumerate.

## altitude→registerTimedReportCallback() YAltitude

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

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| cpp | int **registerTimedReportCallback**( YAltitudeTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YAltitudeTimedReportCallback) **callback** |
| pas | LongInt **registerTimedReportCallback**( **callback**: TYAltitudeTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| uwp | async Task<int> **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| es | async **registerTimedReportCallback**( **callback**) |

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

**Parameters :**

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

## altitude→registerValueCallback() YAltitude

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

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| cpp | int **registerValueCallback**( YAltitudeValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YAltitudeValueCallback) **callback** |
| pas | LongInt **registerValueCallback**( **callback**: TYAltitudeValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| uwp | async Task<int> **registerValueCallback**( ValueCallback **callback**) |
| py | **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| es | async **registerValueCallback**( **callback**) |

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

**Parameters :**

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

## altitude→set_advMode()
## altitude→setAdvMode()

YAltitude

Changes the measuring mode used for the advertised value pushed to the parent hub.

| | |
|---|---|
| js | function **set_advMode**( **newval**) |
| cpp | int **set_advMode**( Y_ADVMODE_enum **newval**) |
| m | -(int) setAdvMode : (Y_ADVMODE_enum) **newval** |
| pas | integer **set_advMode**( **newval**: Integer): integer |
| vb | function **set_advMode**( ByVal **newval** As Integer) As Integer |
| cs | int **set_advMode**( int **newval**) |
| dnp | int **set_advMode**( int **newval**) |
| java | int **set_advMode**( int **newval**) |
| uwp | async Task<int> **set_advMode**( int **newval**) |
| py | **set_advMode**( **newval**) |
| php | function **set_advMode**( **$newval**) |
| es | async **set_advMode**( **newval**) |
| cmd | YAltitude **target set_advMode newval** |

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

**Parameters :**

**newval** a value among `Y_ADVMODE_IMMEDIATE`, `Y_ADVMODE_PERIOD_AVG`, `Y_ADVMODE_PERIOD_MIN` and `Y_ADVMODE_PERIOD_MAX` corresponding to the measuring mode used for the advertised value pushed to the parent hub

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

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

## altitude→set_currentValue()
## altitude→setCurrentValue()

YAltitude

Changes the current estimated altitude.

```
js    function set_currentValue( newval)
cpp   int set_currentValue( double newval)
m     -(int) setCurrentValue : (double) newval
pas   integer set_currentValue( newval: double): integer
vb    function set_currentValue( ByVal newval As Double) As Integer
cs    int set_currentValue( double newval)
dnp   int set_currentValue( double newval)
java  int set_currentValue( double newval)
uwp   async Task<int> set_currentValue( double newval)
py    set_currentValue( newval)
php   function set_currentValue( $newval)
es    async set_currentValue( newval)
cmd   YAltitude target set_currentValue newval
```

This allows one to compensate for ambient pressure variations and to work in relative mode. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**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()

Changes the recorded maximal value observed.

```
js    function set_highestValue( newval)
cpp   int set_highestValue( double newval)
m     -(int) setHighestValue : (double) newval
pas   integer set_highestValue( newval: double): integer
vb    function set_highestValue( ByVal newval As Double) As Integer
cs    int set_highestValue( double newval)
dnp   int set_highestValue( double newval)
java  int set_highestValue( double newval)
uwp   async Task<int> set_highestValue( double newval)
py    set_highestValue( newval)
php   function set_highestValue( $newval)
es    async set_highestValue( newval)
cmd   YAltitude target set_highestValue newval
```

Can be used to reset the value returned by get_lowestValue().

**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()

Changes the datalogger recording frequency for this function.

```
js    function set_logFrequency( newval)
cpp   int set_logFrequency( string newval)
m     -(int) setLogFrequency : (NSString*) newval
pas   integer set_logFrequency( newval: string): integer
vb    function set_logFrequency( ByVal newval As String) As Integer
cs    int set_logFrequency( string newval)
dnp   int set_logFrequency( string newval)
java  int set_logFrequency( String newval)
uwp   async Task<int> set_logFrequency( string newval)
py    set_logFrequency( newval)
php   function set_logFrequency( $newval)
es    async set_logFrequency( newval)
cmd   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". Note that setting the datalogger recording frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**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()
## altitude→setLogicalName()

YAltitude

Changes the logical name of the altimeter.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| cpp | int **set_logicalName**( string **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | integer **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| dnp | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| uwp | async Task<int> **set_logicalName**( string **newval**) |
| py | **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| es | async **set_logicalName**( **newval**) |
| cmd | 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()
## altitude→setLowestValue()

YAltitude

Changes the recorded minimal value observed.

```
js      function set_lowestValue( newval)
cpp     int set_lowestValue( double newval)
m       -(int) setLowestValue : (double) newval
pas     integer set_lowestValue( newval: double): integer
vb      function set_lowestValue( ByVal newval As Double) As Integer
cs      int set_lowestValue( double newval)
dnp     int set_lowestValue( double newval)
java    int set_lowestValue( double newval)
uwp     async Task<int> set_lowestValue( double newval)
py      set_lowestValue( newval)
php     function set_lowestValue( $newval)
es      async set_lowestValue( newval)
cmd     YAltitude target set_lowestValue newval
```

Can be used to reset the value returned by get_lowestValue().

**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()

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

```
js      function set_qnh( newval)
cpp     int set_qnh( double newval)
m       -(int) setQnh : (double) newval
pas     integer set_qnh( newval: double): integer
vb      function set_qnh( ByVal newval As Double) As Integer
cs      int set_qnh( double newval)
dnp     int set_qnh( double newval)
java    int set_qnh( double newval)
uwp     async Task<int> set_qnh( double newval)
py      set_qnh( newval)
php     function set_qnh( $newval)
es      async set_qnh( newval)
cmd     YAltitude target set_qnh newval
```

This enables you to compensate for atmospheric pressure changes due to weather conditions. Applicable to barometric altimeters only. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**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()

Changes the timed value notification frequency for this function.

```
js      function set_reportFrequency( newval)
cpp     int set_reportFrequency( string newval)
m       -(int) setReportFrequency : (NSString*) newval
pas     integer set_reportFrequency( newval: string): integer
vb      function set_reportFrequency( ByVal newval As String) As Integer
cs      int set_reportFrequency( string newval)
dnp     int set_reportFrequency( string newval)
java    int set_reportFrequency( String newval)
uwp     async Task<int> set_reportFrequency( string newval)
py      set_reportFrequency( newval)
php     function set_reportFrequency( $newval)
es      async set_reportFrequency( newval)
cmd     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 (e.g. "4/h"). To disable timed value notifications for this function, use the value "OFF". Note that setting the timed value notification frequency to a greater value than the sensor native sampling frequency is unless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**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()

Changes the resolution of the measured physical values.

| | |
|---|---|
| js | function **set_resolution**( **newval**) |
| cpp | int **set_resolution**( double **newval**) |
| m | -(int) setResolution : (double) **newval** |
| pas | integer **set_resolution**( **newval**: double): integer |
| vb | function **set_resolution**( ByVal **newval** As Double) As Integer |
| cs | int **set_resolution**( double **newval**) |
| dnp | int **set_resolution**( double **newval**) |
| java | int **set_resolution**( double **newval**) |
| uwp | async Task<int> **set_resolution**( double **newval**) |
| py | **set_resolution**( **newval**) |
| php | function **set_resolution**( **$newval**) |
| es | async **set_resolution**( **newval**) |
| cmd | 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. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**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.

## altitude→set_userData()
## altitude→setUserData()

YAltitude

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

```
js      function set_userData( data)
cpp     void set_userData( void * data)
m       -(void) setUserData : (id) data
pas     set_userData( data: Tobject)
vb      procedure set_userData( ByVal data As Object)
cs      void set_userData( object data)
java    void set_userData( Object data)
py      set_userData( data)
php     function set_userData( $data)
es      async set_userData( data)
```

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

**Parameters :**

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

## altitude→startDataLogger() YAltitude

Starts the data logger on the device.

| | |
|---|---|
| js | function **startDataLogger**( ) |
| cpp | int **startDataLogger**( ) |
| m | -(int) **startDataLogger** |
| pas | LongInt **startDataLogger**( ): LongInt |
| vb | function **startDataLogger**( ) As Integer |
| cs | int **startDataLogger**( ) |
| dnp | int **startDataLogger**( ) |
| java | int **startDataLogger**( ) |
| uwp | async Task<int> **startDataLogger**( ) |
| py | **startDataLogger**( ) |
| php | function **startDataLogger**( ) |
| es | async **startDataLogger**( ) |
| cmd | YAltitude **target startDataLogger** |

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

**Returns :**
YAPI_SUCCESS if the call succeeds.

## altitude→stopDataLogger() YAltitude

Stops the datalogger on the device.

| | |
|---|---|
| js | function **stopDataLogger**( ) |
| cpp | int **stopDataLogger**( ) |
| m | -(int) **stopDataLogger** |
| pas | LongInt **stopDataLogger**( ): LongInt |
| vb | function **stopDataLogger**( ) As Integer |
| cs | int **stopDataLogger**( ) |
| dnp | int **stopDataLogger**( ) |
| java | int **stopDataLogger**( ) |
| uwp | async Task<int> **stopDataLogger**( ) |
| py | **stopDataLogger**( ) |
| php | function **stopDataLogger**( ) |
| es | async **stopDataLogger**( ) |
| cmd | YAltitude **target stopDataLogger** |

**Returns :**
YAPI_SUCCESS if the call succeeds.

## altitude→unmuteValueCallbacks() YAltitude

Re-enables the propagation of every new advertised value to the parent hub.

| | |
|---|---|
| js | function **unmuteValueCallbacks**( ) |
| cpp | int **unmuteValueCallbacks**( ) |
| m | -(int) **unmuteValueCallbacks** |
| pas | LongInt **unmuteValueCallbacks**( ): LongInt |
| vb | function **unmuteValueCallbacks**( ) As Integer |
| cs | int **unmuteValueCallbacks**( ) |
| dnp | int **unmuteValueCallbacks**( ) |
| java | int **unmuteValueCallbacks**( ) |
| uwp | async Task<int> **unmuteValueCallbacks**( ) |
| py | **unmuteValueCallbacks**( ) |
| php | function **unmuteValueCallbacks**( ) |
| es | async **unmuteValueCallbacks**( ) |
| cmd | YAltitude **target unmuteValueCallbacks** |

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Returns :**
`YAPI_SUCCESS` when the call succeeds.

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

## altitude→wait_async() YAltitude

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

```
js  function wait_async( callback, context)
es  wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

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

**Returns :**

nothing.

# 22.7. Class YGroundSpeed

Ground speed sensor control interface, available for instance in the Yocto-GPS

The `YGroundSpeed` class allows you to read and configure Yoctopuce ground speed sensors. It inherits from `YSensor` class the core functions to read measurements, to register callback functions, and to access the autonomous datalogger.

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

| | |
|---|---|
| es | in HTML: <script src="../../lib/yocto_groundspeed.js"></script><br>in node.js: require('yoctolib-es2017/yocto_groundspeed.js'); |
| js | <script type='text/javascript' src='yocto_groundspeed.js'></script> |
| cpp | #include "yocto_groundspeed.h" |
| m | #import "yocto_groundspeed.h" |
| pas | uses yocto_groundspeed; |
| vb | yocto_groundspeed.vb |
| cs | yocto_groundspeed.cs |
| dnp | import YoctoProxyAPI.YGroundSpeedProxy |
| java | import com.yoctopuce.YoctoAPI.YGroundSpeed; |
| uwp | import com.yoctopuce.YoctoAPI.YGroundSpeed; |
| py | from yocto_groundspeed import * |
| php | require_once('yocto_groundspeed.php'); |
| vi | YGroundSpeed.vi |

## Global functions

**YGroundSpeed.FindGroundSpeed**(**func**)

Retrieves a ground speed sensor for a given identifier.

**YGroundSpeed.FindGroundSpeedInContext**(**yctx**, **func**)

Retrieves a ground speed sensor for a given identifier in a YAPI context.

**YGroundSpeed.FirstGroundSpeed**()

Starts the enumeration of ground speed sensors currently accessible.

**YGroundSpeed.FirstGroundSpeedInContext**(**yctx**)

Starts the enumeration of ground speed sensors currently accessible.

**YGroundSpeed.GetSimilarFunctions**()

Enumerates all functions of type GroundSpeed available on the devices currently reachable by the library, and returns their unique hardware ID.

## YGroundSpeed properties

**groundspeed→AdvMode** *[writable]*

Measuring mode used for the advertised value pushed to the parent hub.

**groundspeed→AdvertisedValue** *[read-only]*

Short string representing the current state of the function.

**groundspeed→FriendlyName** *[read-only]*

Global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

**groundspeed→FunctionId** *[read-only]*

Hardware identifier of the sensor, without reference to the module.

**groundspeed→HardwareId** *[read-only]*

Unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`.

**groundspeed→IsOnline** *[read-only]*

Checks if the function is currently reachable.

**groundspeed→LogFrequency** *[writable]*

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

**groundspeed→LogicalName** *[writable]*

Logical name of the function.

**groundspeed→ReportFrequency** *[writable]*

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

**groundspeed→Resolution** *[writable]*

Resolution of the measured values.

**groundspeed→SerialNumber** *[read-only]*

Serial number of the module, as set by the factory.

## YGroundSpeed methods

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

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

**groundspeed→clearCache**()

Invalidates the cache.

**groundspeed→describe**()

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

**groundspeed→get_advMode**()

Returns the measuring mode used for the advertised value pushed to the parent hub.

**groundspeed→get_advertisedValue**()

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

**groundspeed→get_currentRawValue**()

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

**groundspeed→get_currentValue**()

Returns the current value of the ground speed, in km/h, as a floating point number.

**groundspeed→get_dataLogger**()

Returns the `YDatalogger` object of the device hosting the sensor.

**groundspeed→get_errorMessage**()

Returns the error message of the latest error with the ground speed sensor.

**groundspeed→get_errorType**()

Returns the numerical error code of the latest error with the ground speed sensor.

**groundspeed→get_friendlyName**()

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

**groundspeed→get_functionDescriptor**()

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

**groundspeed→get_functionId**()

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

**groundspeed→get_hardwareId**()

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

**groundspeed→get_highestValue**()

Returns the maximal value observed for the ground speed since the device was started.

**groundspeed→get_logFrequency()**

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

**groundspeed→get_logicalName()**

Returns the logical name of the ground speed sensor.

**groundspeed→get_lowestValue()**

Returns the minimal value observed for the ground speed since the device was started.

**groundspeed→get_module()**

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

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

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

**groundspeed→get_recordedData(startTime, endTime)**

Retrieves a `YDataSet` object holding historical data for this sensor, for a specified time interval.

**groundspeed→get_reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

**groundspeed→get_resolution()**

Returns the resolution of the measured values.

**groundspeed→get_sensorState()**

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

**groundspeed→get_serialNumber()**

Returns the serial number of the module, as set by the factory.

**groundspeed→get_unit()**

Returns the measuring unit for the ground speed.

**groundspeed→get_userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

**groundspeed→isOnline()**

Checks if the ground speed sensor is currently reachable, without raising any error.

**groundspeed→isOnline_async(callback, context)**

Checks if the ground speed sensor is currently reachable, without raising any error (asynchronous version).

**groundspeed→isReadOnly()**

Test if the function is readOnly.

**groundspeed→isSensorReady()**

Checks if the sensor is currently able to provide an up-to-date measure.

**groundspeed→load(msValidity)**

Preloads the ground speed sensor cache with a specified validity duration.

**groundspeed→loadAttribute(attrName)**

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

**groundspeed→loadCalibrationPoints(rawValues, refValues)**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

**groundspeed→load_async(msValidity, callback, context)**

Preloads the ground speed sensor cache with a specified validity duration (asynchronous version).

**groundspeed→muteValueCallbacks()**

Disables the propagation of every new advertised value to the parent hub.

**groundspeed→nextGroundSpeed**()

Continues the enumeration of ground speed sensors started using `yFirstGroundSpeed()`.

**groundspeed→registerTimedReportCallback**(**callback**)

Registers the callback function that is invoked on every periodic timed notification.

**groundspeed→registerValueCallback**(**callback**)

Registers the callback function that is invoked on every change of advertised value.

**groundspeed→set_advMode**(**newval**)

Changes the measuring mode used for the advertised value pushed to the parent hub.

**groundspeed→set_highestValue**(**newval**)

Changes the recorded maximal value observed.

**groundspeed→set_logFrequency**(**newval**)

Changes the datalogger recording frequency for this function.

**groundspeed→set_logicalName**(**newval**)

Changes the logical name of the ground speed sensor.

**groundspeed→set_lowestValue**(**newval**)

Changes the recorded minimal value observed.

**groundspeed→set_reportFrequency**(**newval**)

Changes the timed value notification frequency for this function.

**groundspeed→set_resolution**(**newval**)

Changes the resolution of the measured physical values.

**groundspeed→set_userData**(**data**)

Stores a user context provided as argument in the userData attribute of the function.

**groundspeed→startDataLogger**()

Starts the data logger on the device.

**groundspeed→stopDataLogger**()

Stops the datalogger on the device.

**groundspeed→unmuteValueCallbacks**()

Re-enables the propagation of every new advertised value to the parent hub.

**groundspeed→wait_async**(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YGroundSpeed.FindGroundSpeed() YGroundSpeed
## YGroundSpeed.FindGroundSpeed()

Retrieves a ground speed sensor for a given identifier.

```
js    function yFindGroundSpeed( func)
cpp   YGroundSpeed* yFindGroundSpeed( string func)
m     +(YGroundSpeed*) FindGroundSpeed : (NSString*) func
pas   TYGroundSpeed yFindGroundSpeed( func: string): TYGroundSpeed
vb    function yFindGroundSpeed( ByVal func As String) As YGroundSpeed
cs    static YGroundSpeed FindGroundSpeed( string func)
dnp   static YGroundSpeedProxy FindGroundSpeed( string func)
java  static YGroundSpeed FindGroundSpeed( String func)
uwp   static YGroundSpeed FindGroundSpeed( string func)
py    FindGroundSpeed( func)
php   function yFindGroundSpeed( $func)
es    static FindGroundSpeed( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the ground speed sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YGroundSpeed.isOnline()` to test if the ground speed sensor is indeed online at a given time. In case of ambiguity when looking for a ground speed sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

**Parameters :**

**func** a string that uniquely characterizes the ground speed sensor, for instance `YGNSSMK1.groundSpeed`.

**Returns :**

a `YGroundSpeed` object allowing you to drive the ground speed sensor.

## YGroundSpeed.FindGroundSpeedInContext() YGroundSpeed
## YGroundSpeed.FindGroundSpeedInContext()

Retrieves a ground speed sensor for a given identifier in a YAPI context.

```
java  static YGroundSpeed FindGroundSpeedInContext( YAPIContext yctx,
                                                    String func)
uwp   static YGroundSpeed FindGroundSpeedInContext( YAPIContext yctx,
                                                    string func)
es    static FindGroundSpeedInContext( yctx, func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the ground speed sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YGroundSpeed.isOnline()` to test if the ground speed sensor is indeed online at a given time. In case of ambiguity when looking for a ground speed sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**yctx** a YAPI context

**func** a string that uniquely characterizes the ground speed sensor, for instance `YGNSSMK1.groundSpeed`.

**Returns :**

a `YGroundSpeed` object allowing you to drive the ground speed sensor.

## YGroundSpeed.FirstGroundSpeed() YGroundSpeed
## YGroundSpeed.FirstGroundSpeed()

Starts the enumeration of ground speed sensors currently accessible.

| | |
|---|---|
| js | function **yFirstGroundSpeed**( ) |
| cpp | YGroundSpeed * **yFirstGroundSpeed**( ) |
| m | +(YGroundSpeed*) **FirstGroundSpeed** |
| pas | TYGroundSpeed **yFirstGroundSpeed**( ): TYGroundSpeed |
| vb | function **yFirstGroundSpeed**( ) As YGroundSpeed |
| cs | static YGroundSpeed **FirstGroundSpeed**( ) |
| java | static YGroundSpeed **FirstGroundSpeed**( ) |
| uwp | static YGroundSpeed **FirstGroundSpeed**( ) |
| py | **FirstGroundSpeed**( ) |
| php | function **yFirstGroundSpeed**( ) |
| es | static **FirstGroundSpeed**( ) |

Use the method `YGroundSpeed.nextGroundSpeed()` to iterate on next ground speed sensors.

**Returns :**

a pointer to a `YGroundSpeed` object, corresponding to the first ground speed sensor currently online, or a `null` pointer if there are none.

## YGroundSpeed.FirstGroundSpeedInContext() YGroundSpeed
## YGroundSpeed.FirstGroundSpeedInContext()

Starts the enumeration of ground speed sensors currently accessible.

java static YGroundSpeed **FirstGroundSpeedInContext**( YAPIContext **yctx**)
uwp static YGroundSpeed **FirstGroundSpeedInContext**( YAPIContext **yctx**)
es static **FirstGroundSpeedInContext**( **yctx**)

Use the method `YGroundSpeed.nextGroundSpeed()` to iterate on next ground speed sensors.

**Parameters :**

**yctx** a YAPI context.

**Returns :**

a pointer to a `YGroundSpeed` object, corresponding to the first ground speed sensor currently online, or a `null` pointer if there are none.

## YGroundSpeed.GetSimilarFunctions() YGroundSpeed
## YGroundSpeed.GetSimilarFunctions()

Enumerates all functions of type GroundSpeed available on the devices currently reachable by the library, and returns their unique hardware ID.

dnp static new string[] **GetSimilarFunctions**( )

Each of these IDs can be provided as argument to the method `YGroundSpeed.FindGroundSpeed` to obtain an object that can control the corresponding device.

**Returns :**
an array of strings, each string containing the unique hardwareId of a device function currently connected.

## groundspeed→AdvMode — YGroundSpeed

Measuring mode used for the advertised value pushed to the parent hub.

dnp int **AdvMode**

**Possible values:**

Y_ADVMODE_INVALID = 0
Y_ADVMODE_IMMEDIATE = 1
Y_ADVMODE_PERIOD_AVG = 2
Y_ADVMODE_PERIOD_MIN = 3
Y_ADVMODE_PERIOD_MAX = 4

**Writable**. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

## groundspeed→AdvertisedValue — YGroundSpeed

Short string representing the current state of the function.

dnp string **AdvertisedValue**

## groundspeed→FriendlyName — YGroundSpeed

Global identifier of the function in the format MODULE_NAME.FUNCTION_NAME.

dnp string **FriendlyName**

The returned string uses the logical names of the module and of the function if they are defined, otherwise the serial number of the module and the hardware identifier of the function (for example: MyCustomName.relay1)

## groundspeed→FunctionId — YGroundSpeed

Hardware identifier of the sensor, without reference to the module.

dnp string **FunctionId**

For example `relay1`

## groundspeed→HardwareId — YGroundSpeed

Unique hardware identifier of the function in the form SERIAL.FUNCTIONID.

dnp string **HardwareId**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example RELAYLO1-123456.relay1).

## groundspeed→IsOnline YGroundSpeed

Checks if the function is currently reachable.

dnp bool **IsOnline**

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the function.

## groundspeed→LogFrequency — YGroundSpeed

Datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

dnp string **LogFrequency**

**Writable**. Changes the datalogger recording frequency for this function. 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". Note that setting the datalogger recording frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

## groundspeed→LogicalName — YGroundSpeed

Logical name of the function.

dnp string **LogicalName**

**Writable**. 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.

## groundspeed→ReportFrequency — YGroundSpeed

Timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

dnp string **ReportFrequency**

**Writable**. Changes the timed value notification frequency for this function. The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (e.g. "4/h"). To disable timed value notifications for this function, use the value "OFF". Note that setting the timed value notification frequency to a greater value than the sensor native sampling frequency is unless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

## groundspeed→Resolution YGroundSpeed

Resolution of the measured values.

dnp double **Resolution**

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Writable**. Changes the resolution of the measured physical values. The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

## groundspeed→SerialNumber — YGroundSpeed

Serial number of the module, as set by the factory.

dnp string **SerialNumber**

## groundspeed→calibrateFromPoints() YGroundSpeed

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```
js    function calibrateFromPoints( rawValues, refValues)
cpp   int calibrateFromPoints( vector<double> rawValues,
                               vector<double> refValues)
m     -(int) calibrateFromPoints : (NSMutableArray*) rawValues
                                 : (NSMutableArray*) refValues
pas   LongInt calibrateFromPoints( rawValues: TDoubleArray,
                                   refValues: TDoubleArray): LongInt
vb    procedure calibrateFromPoints( )
cs    int calibrateFromPoints( List<double> rawValues,
                               List<double> refValues)
dnp   int calibrateFromPoints( )
java  int calibrateFromPoints( ArrayList<Double> rawValues,
                               ArrayList<Double> refValues)
uwp   async Task<int> calibrateFromPoints( List<double> rawValues,
                                           List<double> refValues)
py    calibrateFromPoints( rawValues, refValues)
php   function calibrateFromPoints( $rawValues, $refValues)
es    async calibrateFromPoints( rawValues, refValues)
cmd   YGroundSpeed 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.

## groundspeed→clearCache() YGroundSpeed

Invalidates the cache.

| | |
|---|---|
| js | function **clearCache**( ) |
| cpp | void **clearCache**( ) |
| m | -(void) **clearCache** |
| pas | **clearCache**( ) |
| vb | procedure **clearCache**( ) |
| cs | void **clearCache**( ) |
| java | void **clearCache**( ) |
| py | **clearCache**( ) |
| php | function **clearCache**( ) |
| es | async **clearCache**( ) |

Invalidates the cache of the ground speed sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

## groundspeed→describe() YGroundSpeed

Returns a short text that describes unambiguously the instance of the ground speed sensor in the form TYPE(NAME)=SERIAL.FUNCTIONID.

| | |
|---|---|
| js | function **describe**( ) |
| cpp | string **describe**( ) |
| m | -(NSString*) **describe** |
| pas | string **describe**( ): string |
| vb | function **describe**( ) As String |
| cs | string **describe**( ) |
| java | String **describe**( ) |
| py | **describe**( ) |
| php | function **describe**( ) |
| es | async **describe**( ) |

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**
a string that describes the ground speed sensor (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

## groundspeed→get_advMode()
## groundspeed→advMode()

YGroundSpeed

Returns the measuring mode used for the advertised value pushed to the parent hub.

```
js      function get_advMode( )
cpp     Y_ADVMODE_enum get_advMode( )
m       -(Y_ADVMODE_enum) advMode
pas     Integer get_advMode( ): Integer
vb      function get_advMode( ) As Integer
cs      int get_advMode( )
dnp     int get_advMode( )
java    int get_advMode( )
uwp     async Task<int> get_advMode( )
py      get_advMode( )
php     function get_advMode( )
es      async get_advMode( )
cmd     YGroundSpeed target get_advMode
```

**Returns :**

a value among `Y_ADVMODE_IMMEDIATE`, `Y_ADVMODE_PERIOD_AVG`, `Y_ADVMODE_PERIOD_MIN` and `Y_ADVMODE_PERIOD_MAX` corresponding to the measuring mode used for the advertised value pushed to the parent hub

On failure, throws an exception or returns `Y_ADVMODE_INVALID`.

## groundspeed→get_advertisedValue() YGroundSpeed
## groundspeed→advertisedValue()

Returns the current value of the ground speed sensor (no more than 6 characters).

| | |
|---|---|
| js | function **get_advertisedValue**( ) |
| cpp | string **get_advertisedValue**( ) |
| m | -(NSString*) advertisedValue |
| pas | string **get_advertisedValue**( ): string |
| vb | function **get_advertisedValue**( ) As String |
| cs | string **get_advertisedValue**( ) |
| dnp | string **get_advertisedValue**( ) |
| java | String **get_advertisedValue**( ) |
| uwp | async Task<string> **get_advertisedValue**( ) |
| py | **get_advertisedValue**( ) |
| php | function **get_advertisedValue**( ) |
| es | async **get_advertisedValue**( ) |
| cmd | YGroundSpeed **target get_advertisedValue** |

**Returns :**

a string corresponding to the current value of the ground speed sensor (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

## groundspeed→get_currentRawValue() YGroundSpeed
## groundspeed→currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in km/h, as a floating point number.

```
js      function get_currentRawValue( )
cpp     double get_currentRawValue( )
m       -(double) currentRawValue
pas     double get_currentRawValue( ): double
vb      function get_currentRawValue( ) As Double
cs      double get_currentRawValue( )
dnp     double get_currentRawValue( )
java    double get_currentRawValue( )
uwp     async Task<double> get_currentRawValue( )
py      get_currentRawValue( )
php     function get_currentRawValue( )
es      async get_currentRawValue( )
cmd     YGroundSpeed target get_currentRawValue
```

**Returns :**

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in km/h, as a floating point number

On failure, throws an exception or returns `Y_CURRENTRAWVALUE_INVALID`.

## groundspeed→get_currentValue() YGroundSpeed
## groundspeed→currentValue()

Returns the current value of the ground speed, in km/h, as a floating point number.

```
js      function get_currentValue( )
cpp     double get_currentValue( )
m       -(double) currentValue
pas     double get_currentValue( ): double
vb      function get_currentValue( ) As Double
cs      double get_currentValue( )
dnp     double get_currentValue( )
java    double get_currentValue( )
uwp     async Task<double> get_currentValue( )
py      get_currentValue( )
php     function get_currentValue( )
es      async get_currentValue( )
cmd     YGroundSpeed target get_currentValue
```

Note that a get_currentValue() call will *not* start a measure in the device, it will just return the last measure that occurred in the device. Indeed, internally, each Yoctopuce devices is continuously making measurements at a hardware specific frequency.

If continuously calling get_currentValue() leads you to performances issues, then you might consider to switch to callback programming model. Check the "advanced programming" chapter in in your device user manual for more information.

**Returns :**

a floating point number corresponding to the current value of the ground speed, in km/h, as a floating point number

On failure, throws an exception or returns `Y_CURRENTVALUE_INVALID`.

## groundspeed→get_dataLogger()
## groundspeed→dataLogger()

YGroundSpeed

Returns the `YDatalogger` object of the device hosting the sensor.

| | |
|---|---|
| js | function **get_dataLogger**( ) |
| cpp | YDataLogger* **get_dataLogger**( ) |
| m | -(YDataLogger*) dataLogger |
| pas | TYDataLogger **get_dataLogger**( ): TYDataLogger |
| vb | function **get_dataLogger**( ) As YDataLogger |
| cs | YDataLogger **get_dataLogger**( ) |
| dnp | YDataLoggerProxy **get_dataLogger**( ) |
| java | YDataLogger **get_dataLogger**( ) |
| uwp | async Task<YDataLogger> **get_dataLogger**( ) |
| py | **get_dataLogger**( ) |
| php | function **get_dataLogger**( ) |
| es | async **get_dataLogger**( ) |

This method returns an object that can control global parameters of the data logger. The returned object should not be freed.

**Returns :**
an `YDatalogger` object, or null on error.

## groundspeed→get_errorMessage() YGroundSpeed
## groundspeed→errorMessage()

Returns the error message of the latest error with the ground speed sensor.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | string **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| es | **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a string corresponding to the latest error message that occured while using the ground speed sensor object

## groundspeed→get_errorType()
## groundspeed→errorType()

YGroundSpeed

Returns the numerical error code of the latest error with the ground speed sensor.

```
js      function get_errorType( )
cpp     YRETCODE get_errorType( )
m       -(YRETCODE) errorType
pas     YRETCODE get_errorType( ): YRETCODE
vb      function get_errorType( ) As YRETCODE
cs      YRETCODE get_errorType( )
java    int get_errorType( )
py      get_errorType( )
php     function get_errorType( )
es      get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a number corresponding to the code of the latest error that occurred while using the ground speed sensor object

## groundspeed→get_friendlyName() YGroundSpeed
## groundspeed→friendlyName()

Returns a global identifier of the ground speed sensor in the format MODULE_NAME.FUNCTION_NAME.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| dnp | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| es | async **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the ground speed sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the ground speed sensor (for example: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the ground speed sensor using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

## groundspeed→get_functionDescriptor() YGroundSpeed
## groundspeed→functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | YFUN_DESCR **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| es | async **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**
an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

## groundspeed→get_functionId()
## groundspeed→functionId()

YGroundSpeed

Returns the hardware identifier of the ground speed sensor, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| dnp | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| py | **get_functionId**( ) |
| php | function **get_functionId**( ) |
| es | async **get_functionId**( ) |

For example `relay1`

**Returns :**
a string that identifies the ground speed sensor (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## groundspeed→get_hardwareId() YGroundSpeed
## groundspeed→hardwareId()

Returns the unique hardware identifier of the ground speed sensor in the form SERIAL.FUNCTIONID.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| dnp | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| es | async **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the ground speed sensor (for example RELAYLO1-123456.relay1).

**Returns :**
a string that uniquely identifies the ground speed sensor (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

## groundspeed→get_highestValue() YGroundSpeed
## groundspeed→highestValue()

Returns the maximal value observed for the ground speed since the device was started.

| | |
|---|---|
| js | function **get_highestValue**( ) |
| cpp | double **get_highestValue**( ) |
| m | -(double) highestValue |
| pas | double **get_highestValue**( ): double |
| vb | function **get_highestValue**( ) As Double |
| cs | double **get_highestValue**( ) |
| dnp | double **get_highestValue**( ) |
| java | double **get_highestValue**( ) |
| uwp | async Task<double> **get_highestValue**( ) |
| py | **get_highestValue**( ) |
| php | function **get_highestValue**( ) |
| es | async **get_highestValue**( ) |
| cmd | YGroundSpeed **target get_highestValue** |

Can be reset to an arbitrary value thanks to set_highestValue().

**Returns :**

a floating point number corresponding to the maximal value observed for the ground speed since the device was started

On failure, throws an exception or returns `Y_HIGHESTVALUE_INVALID`.

## groundspeed→get_logFrequency() YGroundSpeed
## groundspeed→logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

```
js      function get_logFrequency( )
cpp     string get_logFrequency( )
m       -(NSString*) logFrequency
pas     string get_logFrequency( ): string
vb      function get_logFrequency( ) As String
cs      string get_logFrequency( )
dnp     string get_logFrequency( )
java    String get_logFrequency( )
uwp     async Task<string> get_logFrequency( )
py      get_logFrequency( )
php     function get_logFrequency( )
es      async get_logFrequency( )
cmd     YGroundSpeed 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`.

## groundspeed→get_logicalName()
## groundspeed→logicalName()

YGroundSpeed

Returns the logical name of the ground speed sensor.

| | |
|---|---|
| js | function **get_logicalName**( ) |
| cpp | string **get_logicalName**( ) |
| m | -(NSString*) logicalName |
| pas | string **get_logicalName**( ): string |
| vb | function **get_logicalName**( ) As String |
| cs | string **get_logicalName**( ) |
| dnp | string **get_logicalName**( ) |
| java | String **get_logicalName**( ) |
| uwp | async Task<string> **get_logicalName**( ) |
| py | **get_logicalName**( ) |
| php | function **get_logicalName**( ) |
| es | async **get_logicalName**( ) |
| cmd | YGroundSpeed **target get_logicalName** |

**Returns :**

a string corresponding to the logical name of the ground speed sensor.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

## groundspeed→get_lowestValue() YGroundSpeed
## groundspeed→lowestValue()

Returns the minimal value observed for the ground speed since the device was started.

```
js      function get_lowestValue( )
cpp     double get_lowestValue( )
m       -(double) lowestValue
pas     double get_lowestValue( ): double
vb      function get_lowestValue( ) As Double
cs      double get_lowestValue( )
dnp     double get_lowestValue( )
java    double get_lowestValue( )
uwp     async Task<double> get_lowestValue( )
py      get_lowestValue( )
php     function get_lowestValue( )
es      async get_lowestValue( )
cmd     YGroundSpeed target get_lowestValue
```

Can be reset to an arbitrary value thanks to set_lowestValue().

**Returns :**

a floating point number corresponding to the minimal value observed for the ground speed since the device was started

On failure, throws an exception or returns `Y_LOWESTVALUE_INVALID`.

## groundspeed→get_module()
## groundspeed→module()

YGroundSpeed

Gets the YModule object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | TYModule **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| dnp | YModuleProxy **get_module**( ) |
| java | YModule **get_module**( ) |
| py | **get_module**( ) |
| php | function **get_module**( ) |
| es | async **get_module**( ) |

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

**Returns :**
an instance of YModule

## groundspeed→get_module_async() YGroundSpeed
## groundspeed→module_async()

Gets the YModule object for the device on which the function is located (asynchronous version).

```
js  function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as on-line.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking Firefox JavaScript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous JavasSript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## groundspeed→get_recordedData() YGroundSpeed
## groundspeed→recordedData()

Retrieves a YDataSet object holding historical data for this sensor, for a specified time interval.

```
js      function get_recordedData( startTime, endTime)
cpp     YDataSet get_recordedData( double startTime, double endTime)
m       -(YDataSet*) recordedData : (double) startTime
                                  : (double) endTime
pas     TYDataSet get_recordedData( startTime: double, endTime: double): TYDataSet
vb      function get_recordedData( ) As YDataSet
cs      YDataSet get_recordedData( double startTime, double endTime)
dnp     YDataSetProxy get_recordedData( double startTime,
                                        double endTime)
java    YDataSet get_recordedData( double startTime, double endTime)
uwp     async Task<YDataSet> get_recordedData( double startTime,
                                               double endTime)
py      get_recordedData( startTime, endTime)
php     function get_recordedData( $startTime, $endTime)
es      async get_recordedData( startTime, endTime)
cmd     YGroundSpeed 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 YDataSet 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 YDataSet 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 measure, 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 measure, 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.

## groundspeed→get_reportFrequency()
## groundspeed→reportFrequency()

YGroundSpeed

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

| | |
|---|---|
| js | function **get_reportFrequency**( ) |
| cpp | string **get_reportFrequency**( ) |
| m | -(NSString*) reportFrequency |
| pas | string **get_reportFrequency**( ): string |
| vb | function **get_reportFrequency**( ) As String |
| cs | string **get_reportFrequency**( ) |
| dnp | string **get_reportFrequency**( ) |
| java | String **get_reportFrequency**( ) |
| uwp | async Task<string> **get_reportFrequency**( ) |
| py | **get_reportFrequency**( ) |
| php | function **get_reportFrequency**( ) |
| es | async **get_reportFrequency**( ) |
| cmd | YGroundSpeed **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`.

## groundspeed→get_resolution()
## groundspeed→resolution()

YGroundSpeed

Returns the resolution of the measured values.

| | |
|---|---|
| js | function **get_resolution**( ) |
| cpp | double **get_resolution**( ) |
| m | -(double) resolution |
| pas | double **get_resolution**( ): double |
| vb | function **get_resolution**( ) As Double |
| cs | double **get_resolution**( ) |
| dnp | double **get_resolution**( ) |
| java | double **get_resolution**( ) |
| uwp | async Task<double> **get_resolution**( ) |
| py | **get_resolution**( ) |
| php | function **get_resolution**( ) |
| es | async **get_resolution**( ) |
| cmd | YGroundSpeed **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. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Returns :**
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

## groundspeed→get_sensorState()
## groundspeed→sensorState()

YGroundSpeed

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

```
js    function get_sensorState( )
cpp   int get_sensorState( )
m     -(int) sensorState
pas   LongInt get_sensorState( ): LongInt
vb    function get_sensorState( ) As Integer
cs    int get_sensorState( )
dnp   int get_sensorState( )
java  int get_sensorState( )
uwp   async Task<int> get_sensorState( )
py    get_sensorState( )
php   function get_sensorState( )
es    async get_sensorState( )
cmd   YGroundSpeed target get_sensorState
```

**Returns :**

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

## groundspeed→get_serialNumber() YGroundSpeed
## groundspeed→serialNumber()

Returns the serial number of the module, as set by the factory.

```
js      function get_serialNumber( )
cpp     string get_serialNumber( )
m       -(NSString*) serialNumber
pas     string get_serialNumber( ): string
vb      function get_serialNumber( ) As String
cs      string get_serialNumber( )
dnp     string get_serialNumber( )
java    String get_serialNumber( )
uwp     async Task<string> get_serialNumber( )
py      get_serialNumber( )
php     function get_serialNumber( )
es      async get_serialNumber( )
cmd     YGroundSpeed 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 YModule.SERIALNUMBER_INVALID.

## groundspeed→get_unit()
## groundspeed→unit()

YGroundSpeed

Returns the measuring unit for the ground speed.

```
js    function get_unit( )
cpp   string get_unit( )
m     -(NSString*) unit
pas   string get_unit( ): string
vb    function get_unit( ) As String
cs    string get_unit( )
dnp   string get_unit( )
java  String get_unit( )
uwp   async Task<string> get_unit( )
py    get_unit( )
php   function get_unit( )
es    async get_unit( )
cmd   YGroundSpeed target get_unit
```

**Returns :**
a string corresponding to the measuring unit for the ground speed

On failure, throws an exception or returns `Y_UNIT_INVALID`.

## groundspeed→get_userData()
## groundspeed→userData()

YGroundSpeed

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(id) userData |
| pas | Tobject **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | **get_userData**( ) |
| php | function **get_userData**( ) |
| es | async **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## groundspeed→isOnline() YGroundSpeed

Checks if the ground speed sensor is currently reachable, without raising any error.

```
js      function isOnline( )
cpp     bool isOnline( )
m       -(BOOL) isOnline
pas     boolean isOnline( ): boolean
vb      function isOnline( ) As Boolean
cs      bool isOnline( )
dnp     bool isOnline( )
java    boolean isOnline( )
py      isOnline( )
php     function isOnline( )
es      async isOnline( )
```

If there is a cached value for the ground speed sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the ground speed sensor.

**Returns :**
`true` if the ground speed sensor can be reached, and `false` otherwise

## groundspeed→isOnline_async() YGroundSpeed

Checks if the ground speed sensor is currently reachable, without raising any error (asynchronous version).

```
js  function isOnline_async( callback, context)
```

If there is a cached value for the ground speed sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## groundspeed→isReadOnly() YGroundSpeed

Test if the function is readOnly.

| | |
|---|---|
| cpp | bool **isReadOnly**( ) |
| m | -(bool) **isReadOnly** |
| pas | boolean **isReadOnly**( ): boolean |
| vb | function **isReadOnly**( ) As Boolean |
| cs | bool **isReadOnly**( ) |
| dnp | bool **isReadOnly**( ) |
| java | boolean **isReadOnly**( ) |
| uwp | async Task<bool> **isReadOnly**( ) |
| py | **isReadOnly**( ) |
| php | function **isReadOnly**( ) |
| es | async **isReadOnly**( ) |
| cmd | YGroundSpeed **target isReadOnly** |

Return `true` if the function is write protected or that the function is not available.

**Returns :**
`true` if the function is readOnly or not online.

## groundspeed→isSensorReady() YGroundSpeed

Checks if the sensor is currently able to provide an up-to-date measure.

```
cmd YGroundSpeed target isSensorReady
```

Returns false if the device is unreachable, or if the sensor does not have a current measure to transmit. No exception is raised if there is an error while trying to contact the device hosting $THEFUNCTION$.

**Returns :**
`true` if the sensor can provide an up-to-date measure, and `false` otherwise

## groundspeed→load() YGroundSpeed

Preloads the ground speed sensor cache with a specified validity duration.

```
js      function load( msValidity)
cpp     YRETCODE load( int msValidity)
m       -(YRETCODE) load : (u64) msValidity
pas     YRETCODE load( msValidity: u64): YRETCODE
vb      function load( ByVal msValidity As Long) As YRETCODE
cs      YRETCODE load( ulong msValidity)
java    int load( long msValidity)
py      load( msValidity)
php     function load( $msValidity)
es      async load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

## groundspeed→loadAttribute() YGroundSpeed

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
js      function loadAttribute( attrName)
cpp     string loadAttribute( string attrName)
m       -(NSString*) loadAttribute : (NSString*) attrName
pas     string loadAttribute( attrName: string): string
vb      function loadAttribute( ) As String
cs      string loadAttribute( string attrName)
dnp     string loadAttribute( string attrName)
java    String loadAttribute( String attrName)
uwp     async Task<string> loadAttribute( string attrName)
py      loadAttribute( attrName)
php     function loadAttribute( $attrName)
es      async loadAttribute( attrName)
```

**Parameters :**

**attrName** the name of the requested attribute

**Returns :**

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

## groundspeed→loadCalibrationPoints() YGroundSpeed

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
js    function loadCalibrationPoints( rawValues, refValues)
cpp   int loadCalibrationPoints( vector<double> rawValues,
                                 vector<double> refValues)
m     -(int) loadCalibrationPoints : (NSMutableArray*) rawValues
                                   : (NSMutableArray*) refValues
pas   LongInt loadCalibrationPoints( var rawValues: TDoubleArray,
                                     var refValues: TDoubleArray): LongInt
vb    procedure loadCalibrationPoints( )
cs    int loadCalibrationPoints( List<double> rawValues,
                                 List<double> refValues)
dnp   int loadCalibrationPoints( )
java  int loadCalibrationPoints( ArrayList<Double> rawValues,
                                 ArrayList<Double> refValues)
uwp   async Task<int> loadCalibrationPoints( List<double> rawValues,
                                             List<double> refValues)
py    loadCalibrationPoints( rawValues, refValues)
php   function loadCalibrationPoints( &$rawValues, &$refValues)
es    async loadCalibrationPoints( rawValues, refValues)
cmd   YGroundSpeed 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.

## groundspeed→load_async() YGroundSpeed

Preloads the ground speed sensor cache with a specified validity duration (asynchronous version).

```
js  function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking the JavaScript virtual machine.

**Parameters :**

| | |
|---|---|
| **msValidity** | an integer corresponding to the validity of the loaded function parameters, in milliseconds |
| **callback** | callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or YAPI_SUCCESS) |
| **context** | caller-specific object that is passed as-is to the callback function |

**Returns :**

nothing : the result is provided to the callback.

## groundspeed→muteValueCallbacks() YGroundSpeed

Disables the propagation of every new advertised value to the parent hub.

```
js      function muteValueCallbacks( )
cpp     int muteValueCallbacks( )
m       -(int) muteValueCallbacks
pas     LongInt muteValueCallbacks( ): LongInt
vb      function muteValueCallbacks( ) As Integer
cs      int muteValueCallbacks( )
dnp     int muteValueCallbacks( )
java    int muteValueCallbacks( )
uwp     async Task<int> muteValueCallbacks( )
py      muteValueCallbacks( )
php     function muteValueCallbacks( )
es      async muteValueCallbacks( )
cmd     YGroundSpeed target muteValueCallbacks
```

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Returns :**
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

## groundspeed→nextGroundSpeed() YGroundSpeed

Continues the enumeration of ground speed sensors started using `yFirstGroundSpeed()`.

| | |
|---|---|
| js | function **nextGroundSpeed**( ) |
| cpp | YGroundSpeed * **nextGroundSpeed**( ) |
| m | -(YGroundSpeed*) **nextGroundSpeed** |
| pas | TYGroundSpeed **nextGroundSpeed**( ): TYGroundSpeed |
| vb | function **nextGroundSpeed**( ) As YGroundSpeed |
| cs | YGroundSpeed **nextGroundSpeed**( ) |
| java | YGroundSpeed **nextGroundSpeed**( ) |
| uwp | YGroundSpeed **nextGroundSpeed**( ) |
| py | **nextGroundSpeed**( ) |
| php | function **nextGroundSpeed**( ) |
| es | **nextGroundSpeed**( ) |

Caution: You can't make any assumption about the returned ground speed sensors order. If you want to find a specific a ground speed sensor, use `GroundSpeed.findGroundSpeed()` and a hardwareID or a logical name.

**Returns :**

a pointer to a `YGroundSpeed` object, corresponding to a ground speed sensor currently online, or a `null` pointer if there are no more ground speed sensors to enumerate.

## groundspeed→registerTimedReportCallback() YGroundSpeed

Registers the callback function that is invoked on every periodic timed notification.

| | |
|---|---|
| js | function **registerTimedReportCallback**( **callback**) |
| cpp | int **registerTimedReportCallback**( YGroundSpeedTimedReportCallback **callback**) |
| m | -(int) **registerTimedReportCallback** : (YGroundSpeedTimedReportCallback) **callback** |
| pas | LongInt **registerTimedReportCallback**( **callback**: TYGroundSpeedTimedReportCallback): LongInt |
| vb | function **registerTimedReportCallback**( ) As Integer |
| cs | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| java | int **registerTimedReportCallback**( TimedReportCallback **callback**) |
| uwp | async Task<int> **registerTimedReportCallback**( TimedReportCallback **callback**) |
| py | **registerTimedReportCallback**( **callback**) |
| php | function **registerTimedReportCallback**( **$callback**) |
| es | async **registerTimedReportCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

## groundspeed→registerValueCallback() YGroundSpeed

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| cpp | int **registerValueCallback**( YGroundSpeedValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YGroundSpeedValueCallback) **callback** |
| pas | LongInt **registerValueCallback**( **callback**: TYGroundSpeedValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| uwp | async Task<int> **registerValueCallback**( ValueCallback **callback**) |
| py | **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| es | async **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## groundspeed→set_advMode()
## groundspeed→setAdvMode()

YGroundSpeed

Changes the measuring mode used for the advertised value pushed to the parent hub.

| | |
|---|---|
| js | function **set_advMode**( **newval**) |
| cpp | int **set_advMode**( Y_ADVMODE_enum **newval**) |
| m | -(int) setAdvMode : (Y_ADVMODE_enum) **newval** |
| pas | integer **set_advMode**( **newval**: Integer): integer |
| vb | function **set_advMode**( ByVal **newval** As Integer) As Integer |
| cs | int **set_advMode**( int **newval**) |
| dnp | int **set_advMode**( int **newval**) |
| java | int **set_advMode**( int **newval**) |
| uwp | async Task<int> **set_advMode**( int **newval**) |
| py | **set_advMode**( **newval**) |
| php | function **set_advMode**( **$newval**) |
| es | async **set_advMode**( **newval**) |
| cmd | YGroundSpeed **target set_advMode newval** |

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Parameters :**

**newval** a value among `Y_ADVMODE_IMMEDIATE`, `Y_ADVMODE_PERIOD_AVG`, `Y_ADVMODE_PERIOD_MIN` and `Y_ADVMODE_PERIOD_MAX` corresponding to the measuring mode used for the advertised value pushed to the parent hub

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## groundspeed→set_highestValue() YGroundSpeed
## groundspeed→setHighestValue()

Changes the recorded maximal value observed.

| | |
|---|---|
| js | function **set_highestValue**( **newval**) |
| cpp | int **set_highestValue**( double **newval**) |
| m | -(int) setHighestValue : (double) **newval** |
| pas | integer **set_highestValue**( **newval**: double): integer |
| vb | function **set_highestValue**( ByVal **newval** As Double) As Integer |
| cs | int **set_highestValue**( double **newval**) |
| dnp | int **set_highestValue**( double **newval**) |
| java | int **set_highestValue**( double **newval**) |
| uwp | async Task<int> **set_highestValue**( double **newval**) |
| py | **set_highestValue**( **newval**) |
| php | function **set_highestValue**( **$newval**) |
| es | async **set_highestValue**( **newval**) |
| cmd | YGroundSpeed **target set_highestValue newval** |

Can be used to reset the value returned by get_lowestValue().

**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.

## groundspeed→set_logFrequency() YGroundSpeed
## groundspeed→setLogFrequency()

Changes the datalogger recording frequency for this function.

```
js    function set_logFrequency( newval)
cpp   int set_logFrequency( string newval)
m     -(int) setLogFrequency : (NSString*) newval
pas   integer set_logFrequency( newval: string): integer
vb    function set_logFrequency( ByVal newval As String) As Integer
cs    int set_logFrequency( string newval)
dnp   int set_logFrequency( string newval)
java  int set_logFrequency( String newval)
uwp   async Task<int> set_logFrequency( string newval)
py    set_logFrequency( newval)
php   function set_logFrequency( $newval)
es    async set_logFrequency( newval)
cmd   YGroundSpeed 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". Note that setting the datalogger recording frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**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.

## groundspeed→set_logicalName() YGroundSpeed
## groundspeed→setLogicalName()

Changes the logical name of the ground speed sensor.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| cpp | int **set_logicalName**( string **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | integer **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| dnp | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| uwp | async Task<int> **set_logicalName**( string **newval**) |
| py | **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| es | async **set_logicalName**( **newval**) |
| cmd | YGroundSpeed **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 ground speed sensor.

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## groundspeed→set_lowestValue() YGroundSpeed
## groundspeed→setLowestValue()

Changes the recorded minimal value observed.

```
js      function set_lowestValue( newval)
cpp     int set_lowestValue( double newval)
m       -(int) setLowestValue : (double) newval
pas     integer set_lowestValue( newval: double): integer
vb      function set_lowestValue( ByVal newval As Double) As Integer
cs      int set_lowestValue( double newval)
dnp     int set_lowestValue( double newval)
java    int set_lowestValue( double newval)
uwp     async Task<int> set_lowestValue( double newval)
py      set_lowestValue( newval)
php     function set_lowestValue( $newval)
es      async set_lowestValue( newval)
cmd     YGroundSpeed target set_lowestValue newval
```

Can be used to reset the value returned by get_lowestValue().

**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.

## groundspeed→set_reportFrequency() YGroundSpeed
## groundspeed→setReportFrequency()

Changes the timed value notification frequency for this function.

```
js    function set_reportFrequency( newval)
cpp   int set_reportFrequency( string newval)
m     -(int) setReportFrequency : (NSString*) newval
pas   integer set_reportFrequency( newval: string): integer
vb    function set_reportFrequency( ByVal newval As String) As Integer
cs    int set_reportFrequency( string newval)
dnp   int set_reportFrequency( string newval)
java  int set_reportFrequency( String newval)
uwp   async Task<int> set_reportFrequency( string newval)
py    set_reportFrequency( newval)
php   function set_reportFrequency( $newval)
es    async set_reportFrequency( newval)
cmd   YGroundSpeed 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 (e.g. "4/h"). To disable timed value notifications for this function, use the value "OFF". Note that setting the timed value notification frequency to a greater value than the sensor native sampling frequency is unless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**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.

## groundspeed→set_resolution()
## groundspeed→setResolution()

YGroundSpeed

Changes the resolution of the measured physical values.

| | |
|---|---|
| js | function **set_resolution**( **newval**) |
| cpp | int **set_resolution**( double **newval**) |
| m | -(int) setResolution : (double) **newval** |
| pas | integer **set_resolution**( **newval**: double): integer |
| vb | function **set_resolution**( ByVal **newval** As Double) As Integer |
| cs | int **set_resolution**( double **newval**) |
| dnp | int **set_resolution**( double **newval**) |
| java | int **set_resolution**( double **newval**) |
| uwp | async Task<int> **set_resolution**( double **newval**) |
| py | **set_resolution**( **newval**) |
| php | function **set_resolution**( **$newval**) |
| es | async **set_resolution**( **newval**) |
| cmd | YGroundSpeed **target set_resolution newval** |

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**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.

## groundspeed→set_userData()
## groundspeed→setUserData()

YGroundSpeed

Stores a user context provided as argument in the userData attribute of the function.

```
js      function set_userData( data)
cpp     void set_userData( void * data)
m       -(void) setUserData : (id) data
pas     set_userData( data: Tobject)
vb      procedure set_userData( ByVal data As Object)
cs      void set_userData( object data)
java    void set_userData( Object data)
py      set_userData( data)
php     function set_userData( $data)
es      async set_userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## groundspeed→startDataLogger() YGroundSpeed

Starts the data logger on the device.

| | |
|---|---|
| js | function **startDataLogger**( ) |
| cpp | int **startDataLogger**( ) |
| m | -(int) **startDataLogger** |
| pas | LongInt **startDataLogger**( ): LongInt |
| vb | function **startDataLogger**( ) As Integer |
| cs | int **startDataLogger**( ) |
| dnp | int **startDataLogger**( ) |
| java | int **startDataLogger**( ) |
| uwp | async Task<int> **startDataLogger**( ) |
| py | **startDataLogger**( ) |
| php | function **startDataLogger**( ) |
| es | async **startDataLogger**( ) |
| cmd | YGroundSpeed **target** **startDataLogger** |

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

**Returns :**
YAPI_SUCCESS if the call succeeds.

## groundspeed→stopDataLogger() YGroundSpeed

Stops the datalogger on the device.

| | |
|---|---|
| js | function **stopDataLogger**( ) |
| cpp | int **stopDataLogger**( ) |
| m | -(int) **stopDataLogger** |
| pas | LongInt **stopDataLogger**( ): LongInt |
| vb | function **stopDataLogger**( ) As Integer |
| cs | int **stopDataLogger**( ) |
| dnp | int **stopDataLogger**( ) |
| java | int **stopDataLogger**( ) |
| uwp | async Task<int> **stopDataLogger**( ) |
| py | **stopDataLogger**( ) |
| php | function **stopDataLogger**( ) |
| es | async **stopDataLogger**( ) |
| cmd | YGroundSpeed **target stopDataLogger** |

**Returns :**
YAPI_SUCCESS if the call succeeds.

## groundspeed→unmuteValueCallbacks() YGroundSpeed

Re-enables the propagation of every new advertised value to the parent hub.

```
js      function unmuteValueCallbacks( )
cpp     int unmuteValueCallbacks( )
m       -(int) unmuteValueCallbacks
pas     LongInt unmuteValueCallbacks( ): LongInt
vb      function unmuteValueCallbacks( ) As Integer
cs      int unmuteValueCallbacks( )
dnp     int unmuteValueCallbacks( )
java    int unmuteValueCallbacks( )
uwp     async Task<int> unmuteValueCallbacks( )
py      unmuteValueCallbacks( )
php     function unmuteValueCallbacks( )
es      async unmuteValueCallbacks( )
cmd     YGroundSpeed target unmuteValueCallbacks
```

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Returns :**
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

## groundspeed→wait_async() YGroundSpeed

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js  function wait_async( callback, context)
es  wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 22.8. Class YDataLogger

DataLogger control interface, available on most Yoctopuce sensors.

A non-volatile memory for storing ongoing measured data is available on most Yoctopuce sensors. Recording can happen automatically, without requiring a permanent connection to a computer. The `YDataLogger` class controls the global parameters of the internal data logger. Recording control (start/stop) as well as data retreival is done at sensor objects level.

In order to use the functions described here, you should include:

| | |
|---|---|
| js | <script type='text/javascript' src='yocto_module.js'></script> |
| cpp | #include "yocto_module.h" |
| m | #import "yocto_module.h" |
| pas | uses yocto_module; |
| vb | yocto_module.vb |
| cs | yocto_module.cs |
| dnp | import YoctoProxyAPI.YDataLoggerProxy |
| java | import com.yoctopuce.YoctoAPI.YDataLogger; |
| uwp | import com.yoctopuce.YoctoAPI.YDataLogger; |
| py | from yocto_module import * |
| php | require_once('yocto_module.php'); |
| es | in HTML: <script src="../../lib/yocto_module.js"></script><br>in node.js: require('yoctolib-es2017/yocto_module.js'); |
| vi | YDataLogger.vi |

## Global functions

**YDataLogger.FindDataLogger**(**func**)

Retrieves a data logger for a given identifier.

**YDataLogger.FindDataLoggerInContext**(**yctx**, **func**)

Retrieves a data logger for a given identifier in a YAPI context.

**YDataLogger.FirstDataLogger**()

Starts the enumeration of data loggers currently accessible.

**YDataLogger.FirstDataLoggerInContext**(**yctx**)

Starts the enumeration of data loggers currently accessible.

**YDataLogger.GetSimilarFunctions**()

Enumerates all functions of type DataLogger available on the devices currently reachable by the library, and returns their unique hardware ID.

## `YDataLogger` properties

**datalogger→AdvertisedValue** *[read-only]*

Short string representing the current state of the function.

**datalogger→AutoStart** *[writable]*

Default activation state of the data logger on power up.

**datalogger→BeaconDriven** *[writable]*

True if the data logger is synchronised with the localization beacon.

**datalogger→FriendlyName** *[read-only]*

Global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

**datalogger→FunctionId** *[read-only]*

Hardware identifier of the data logger, without reference to the module.

**datalogger→HardwareId** *[read-only]*

Unique hardware identifier of the function in the form SERIAL.FUNCTIONID.

**datalogger→IsOnline** *[read-only]*

Checks if the function is currently reachable.

**datalogger→LogicalName** *[writable]*

Logical name of the function.

**datalogger→Recording** *[writable]*

Current activation state of the data logger.

**datalogger→SerialNumber** *[read-only]*

Serial number of the module, as set by the factory.

### YDataLogger methods

**datalogger→clearCache**()

Invalidates the cache.

**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**()

Returns 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_serialNumber**()

Returns the serial number of the module, as set by the factory.

**datalogger→get_timeUTC**()

Returns the Unix timestamp for current UTC time, if known.

**datalogger→get_usage**()

Returns the percentage of datalogger memory in use.

**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→isReadOnly**()

Test if the function is readOnly.

**datalogger→load**(**msValidity**)

Preloads the data logger cache with a specified validity duration.

**datalogger→loadAttribute**(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

**datalogger→load_async**(**msValidity**, **callback**, **context**)

Preloads the data logger cache with a specified validity duration (asynchronous version).

**datalogger→muteValueCallbacks**()

Disables the propagation of every new advertised value to the parent hub.

**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→unmuteValueCallbacks**()

Re-enables the propagation of every new advertised value to the parent hub.

**datalogger→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

## YDataLogger.FindDataLogger() YDataLogger
## YDataLogger.FindDataLogger()

Retrieves a data logger for a given identifier.

```
js      function yFindDataLogger( func)
cpp     YDataLogger* yFindDataLogger( string func)
m       +(YDataLogger*) FindDataLogger : (NSString*) func
pas     TYDataLogger yFindDataLogger( func: string): TYDataLogger
vb      function yFindDataLogger( ByVal func As String) As YDataLogger
cs      static YDataLogger FindDataLogger( string func)
dnp     static YDataLoggerProxy FindDataLogger( string func)
java    static YDataLogger FindDataLogger( String func)
uwp     static YDataLogger FindDataLogger( string func)
py      FindDataLogger( func)
php     function yFindDataLogger( $func)
es      static FindDataLogger( func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the data logger is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDataLogger.isOnline()` to test if the data logger is indeed online at a given time. In case of ambiguity when looking for a data logger by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

**Parameters :**

**func** a string that uniquely characterizes the data logger, for instance `Y3DMK002.dataLogger`.

**Returns :**

a `YDataLogger` object allowing you to drive the data logger.

## YDataLogger.FindDataLoggerInContext() YDataLogger
## YDataLogger.FindDataLoggerInContext()

Retrieves a data logger for a given identifier in a YAPI context.

```
java  static YDataLogger FindDataLoggerInContext( YAPIContext yctx,
                                                  String func)
uwp   static YDataLogger FindDataLoggerInContext( YAPIContext yctx,
                                                  string func)
es    static FindDataLoggerInContext( yctx, func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the data logger is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDataLogger.isOnline()` to test if the data logger is indeed online at a given time. In case of ambiguity when looking for a data logger by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

**Parameters :**

**yctx** a YAPI context

**func** a string that uniquely characterizes the data logger, for instance `Y3DMK002.dataLogger`.

**Returns :**

a `YDataLogger` object allowing you to drive the data logger.

## YDataLogger.FirstDataLogger() YDataLogger
## YDataLogger.FirstDataLogger()

Starts the enumeration of data loggers currently accessible.

| | |
|---|---|
| js | function **yFirstDataLogger**( ) |
| cpp | YDataLogger * **yFirstDataLogger**( ) |
| m | +(YDataLogger*) **FirstDataLogger** |
| pas | TYDataLogger **yFirstDataLogger**( ): TYDataLogger |
| vb | function **yFirstDataLogger**( ) As YDataLogger |
| cs | static YDataLogger **FirstDataLogger**( ) |
| java | static YDataLogger **FirstDataLogger**( ) |
| uwp | static YDataLogger **FirstDataLogger**( ) |
| py | **FirstDataLogger**( ) |
| php | function **yFirstDataLogger**( ) |
| es | static **FirstDataLogger**( ) |

Use the method `YDataLogger.nextDataLogger()` to iterate on next data loggers.

**Returns :**

a pointer to a `YDataLogger` object, corresponding to the first data logger currently online, or a `null` pointer if there are none.

## YDataLogger.FirstDataLoggerInContext() YDataLogger
## YDataLogger.FirstDataLoggerInContext()

Starts the enumeration of data loggers currently accessible.

| | |
|---|---|
| java | static YDataLogger **FirstDataLoggerInContext**( YAPIContext **yctx**) |
| uwp | static YDataLogger **FirstDataLoggerInContext**( YAPIContext **yctx**) |
| es | static **FirstDataLoggerInContext**( **yctx**) |

Use the method `YDataLogger.nextDataLogger()` to iterate on next data loggers.

**Parameters :**

**yctx** a YAPI context.

**Returns :**

a pointer to a `YDataLogger` object, corresponding to the first data logger currently online, or a `null` pointer if there are none.

## YDataLogger.GetSimilarFunctions() YDataLogger
## YDataLogger.GetSimilarFunctions()

Enumerates all functions of type DataLogger available on the devices currently reachable by the library, and returns their unique hardware ID.

dnp static new string[] **GetSimilarFunctions**( )

Each of these IDs can be provided as argument to the method `YDataLogger.FindDataLogger` to obtain an object that can control the corresponding device.

**Returns :**
an array of strings, each string containing the unique hardwareId of a device function currently connected.

## datalogger→AdvertisedValue — YDataLogger

Short string representing the current state of the function.

dnp string **AdvertisedValue**

## datalogger→AutoStart — YDataLogger

Default activation state of the data logger on power up.

dnp int **AutoStart**

**Possible values:**

| | |
|---|---|
| Y_AUTOSTART_INVALID | = 0 |
| Y_AUTOSTART_OFF | = 1 |
| Y_AUTOSTART_ON | = 2 |

**Writable**. Do not forget to call the `saveToFlash()` method of the module to save the configuration change. Note: if the device doesn't have any time source at his disposal when starting up, it will wait for ~8 seconds before automatically starting to record with an arbitrary timestamp

## datalogger→BeaconDriven — YDataLogger

True if the data logger is synchronised with the localization beacon.

dnp int **BeaconDriven**

**Possible values:**

```
Y_BEACONDRIVEN_INVALID = 0
Y_BEACONDRIVEN_OFF     = 1
Y_BEACONDRIVEN_ON      = 2
```

**Writable**. Changes the type of synchronisation of the data logger. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

## datalogger→FriendlyName — YDataLogger

Global identifier of the function in the format MODULE_NAME.FUNCTION_NAME.

dnp string **FriendlyName**

The returned string uses the logical names of the module and of the function if they are defined, otherwise the serial number of the module and the hardware identifier of the function (for example: MyCustomName.relay1)

## datalogger→FunctionId — YDataLogger

Hardware identifier of the data logger, without reference to the module.

dnp string **FunctionId**

For example `relay1`

## datalogger→HardwareId — YDataLogger

Unique hardware identifier of the function in the form SERIAL.FUNCTIONID.

dnp string **HardwareId**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example RELAYLO1-123456.relay1).

## datalogger→IsOnline YDataLogger

Checks if the function is currently reachable.

dnp bool **IsOnline**

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the function.

## datalogger→LogicalName — YDataLogger

Logical name of the function.

dnp string **LogicalName**

**Writable**. 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.

## datalogger→Recording YDataLogger

Current activation state of the data logger.

dnp int **Recording**

**Possible values:**

```
Y_RECORDING_INVALID = 0
Y_RECORDING_OFF     = 1
Y_RECORDING_ON      = 2
Y_RECORDING_PENDING = 3
```

**Writable**. Changes the activation state of the data logger to start/stop recording data.

## datalogger→SerialNumber — YDataLogger

Serial number of the module, as set by the factory.

dnp string **SerialNumber**

## datalogger→clearCache() YDataLogger

Invalidates the cache.

| | |
|---|---|
| js | function **clearCache**( ) |
| cpp | void **clearCache**( ) |
| m | -(void) **clearCache** |
| pas | **clearCache**( ) |
| vb | procedure **clearCache**( ) |
| cs | void **clearCache**( ) |
| java | void **clearCache**( ) |
| py | **clearCache**( ) |
| php | function **clearCache**( ) |
| es | async **clearCache**( ) |

Invalidates the cache of the data logger attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

## datalogger→describe() YDataLogger

Returns a short text that describes unambiguously the instance of the data logger in the form TYPE(NAME)=SERIAL.FUNCTIONID.

```
js      function describe( )
cpp     string describe( )
m       -(NSString*) describe
pas     string describe( ): string
vb      function describe( ) As String
cs      string describe( )
java    String describe( )
py      describe( )
php     function describe( )
es      async describe( )
```

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

**Returns :**
a string that describes the data logger (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

## datalogger→forgetAllDataStreams() YDataLogger

Clears the data logger memory and discards all recorded data streams.

```
js    function forgetAllDataStreams( )
cpp   int forgetAllDataStreams( )
m     -(int) forgetAllDataStreams
pas   LongInt forgetAllDataStreams( ): LongInt
vb    function forgetAllDataStreams( ) As Integer
cs    int forgetAllDataStreams( )
dnp   int forgetAllDataStreams( )
java  int forgetAllDataStreams( )
uwp   async Task<int> forgetAllDataStreams( )
py    forgetAllDataStreams( )
php   function forgetAllDataStreams( )
es    async forgetAllDataStreams( )
cmd   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()
## datalogger→advertisedValue()

YDataLogger

Returns the current value of the data logger (no more than 6 characters).

```
js      function get_advertisedValue( )
cpp     string get_advertisedValue( )
m       -(NSString*) advertisedValue
pas     string get_advertisedValue( ): string
vb      function get_advertisedValue( ) As String
cs      string get_advertisedValue( )
dnp     string get_advertisedValue( )
java    String get_advertisedValue( )
uwp     async Task<string> get_advertisedValue( )
py      get_advertisedValue( )
php     function get_advertisedValue( )
es      async get_advertisedValue( )
cmd     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()
## datalogger→autoStart()

YDataLogger

Returns the default activation state of the data logger on power up.

| | |
|---|---|
| js | function **get_autoStart**( ) |
| cpp | Y_AUTOSTART_enum **get_autoStart**( ) |
| m | -(Y_AUTOSTART_enum) autoStart |
| pas | Integer **get_autoStart**( ): Integer |
| vb | function **get_autoStart**( ) As Integer |
| cs | int **get_autoStart**( ) |
| dnp | int **get_autoStart**( ) |
| java | int **get_autoStart**( ) |
| uwp | async Task<int> **get_autoStart**( ) |
| py | **get_autoStart**( ) |
| php | function **get_autoStart**( ) |
| es | async **get_autoStart**( ) |
| cmd | 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()
## datalogger→beaconDriven()

YDataLogger

Returns true if the data logger is synchronised with the localization beacon.

```
js      function get_beaconDriven( )
cpp     Y_BEACONDRIVEN_enum get_beaconDriven( )
m       -(Y_BEACONDRIVEN_enum) beaconDriven
pas     Integer get_beaconDriven( ): Integer
vb      function get_beaconDriven( ) As Integer
cs      int get_beaconDriven( )
dnp     int get_beaconDriven( )
java    int get_beaconDriven( )
uwp     async Task<int> get_beaconDriven( )
py      get_beaconDriven( )
php     function get_beaconDriven( )
es      async get_beaconDriven( )
cmd     YDataLogger target get_beaconDriven
```

**Returns :**

either `Y_BEACONDRIVEN_OFF` or `Y_BEACONDRIVEN_ON`, according to true if the data logger is synchronised with the localization beacon

On failure, throws an exception or returns `Y_BEACONDRIVEN_INVALID`.

## datalogger→get_currentRunIndex() YDataLogger
## datalogger→currentRunIndex()

Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.

```
js      function get_currentRunIndex( )
cpp     int get_currentRunIndex( )
m       -(int) currentRunIndex
pas     LongInt get_currentRunIndex( ): LongInt
vb      function get_currentRunIndex( ) As Integer
cs      int get_currentRunIndex( )
dnp     int get_currentRunIndex( )
java    int get_currentRunIndex( )
uwp     async Task<int> get_currentRunIndex( )
py      get_currentRunIndex( )
php     function get_currentRunIndex( )
es      async get_currentRunIndex( )
cmd     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()
## datalogger→dataSets()

YDataLogger

Returns a list of `YDataSet` objects that can be used to retrieve all measures stored by the data logger.

| | |
|---|---|
| js | function **get_dataSets**( ) |
| cpp | vector<YDataSet> **get_dataSets**( ) |
| m | -(NSMutableArray*) dataSets |
| pas | TYDataSetArray **get_dataSets**( ): TYDataSetArray |
| vb | function **get_dataSets**( ) As List |
| cs | List<YDataSet> **get_dataSets**( ) |
| dnp | YDataSetProxy[] **get_dataSets**( ) |
| java | ArrayList<YDataSet> **get_dataSets**( ) |
| uwp | async Task<List<YDataSet>> **get_dataSets**( ) |
| py | **get_dataSets**( ) |
| php | function **get_dataSets**( ) |
| es | async **get_dataSets**( ) |
| cmd | 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_dataStreams() YDataLogger
## datalogger→dataStreams()

Builds a list of all data streams hold by the data logger (legacy method).

The caller must pass by reference an empty array to hold `DataStream` objects, and the function fills it with objects describing available data sequences.

This is the old way to retrieve data from the DataLogger. For new applications, you should rather use `get_dataSets()` method, or call directly `get_recordedData()` on the sensor object.

**Parameters :**

**v** an array of `DataStream` objects to be filled in

**Returns :**

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

## datalogger→get_errorMessage()
## datalogger→errorMessage()

YDataLogger

Returns the error message of the latest error with the data logger.

| | |
|---|---|
| js | function **get_errorMessage**( ) |
| cpp | string **get_errorMessage**( ) |
| m | -(NSString*) errorMessage |
| pas | string **get_errorMessage**( ): string |
| vb | function **get_errorMessage**( ) As String |
| cs | string **get_errorMessage**( ) |
| java | String **get_errorMessage**( ) |
| py | **get_errorMessage**( ) |
| php | function **get_errorMessage**( ) |
| es | **get_errorMessage**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a string corresponding to the latest error message that occured while using the data logger object

## datalogger→get_errorType()
## datalogger→errorType()

YDataLogger

Returns the numerical error code of the latest error with the data logger.

| | |
|---|---|
| js | function **get_errorType**( ) |
| cpp | YRETCODE **get_errorType**( ) |
| m | -(YRETCODE) errorType |
| pas | YRETCODE **get_errorType**( ): YRETCODE |
| vb | function **get_errorType**( ) As YRETCODE |
| cs | YRETCODE **get_errorType**( ) |
| java | int **get_errorType**( ) |
| py | **get_errorType**( ) |
| php | function **get_errorType**( ) |
| es | **get_errorType**( ) |

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

**Returns :**
a number corresponding to the code of the latest error that occurred while using the data logger object

## datalogger→get_friendlyName()
## datalogger→friendlyName()

YDataLogger

Returns a global identifier of the data logger in the format MODULE_NAME.FUNCTION_NAME.

| | |
|---|---|
| js | function **get_friendlyName**( ) |
| cpp | string **get_friendlyName**( ) |
| m | -(NSString*) friendlyName |
| cs | string **get_friendlyName**( ) |
| dnp | string **get_friendlyName**( ) |
| java | String **get_friendlyName**( ) |
| py | **get_friendlyName**( ) |
| php | function **get_friendlyName**( ) |
| es | async **get_friendlyName**( ) |

The returned string uses the logical names of the module and of the data logger if they are defined, otherwise the serial number of the module and the hardware identifier of the data logger (for example: MyCustomName.relay1)

**Returns :**

a string that uniquely identifies the data logger using logical names (ex: MyCustomName.relay1)

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

## datalogger→get_functionDescriptor() YDataLogger
## datalogger→functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

| | |
|---|---|
| js | function **get_functionDescriptor**( ) |
| cpp | YFUN_DESCR **get_functionDescriptor**( ) |
| m | -(YFUN_DESCR) functionDescriptor |
| pas | YFUN_DESCR **get_functionDescriptor**( ): YFUN_DESCR |
| vb | function **get_functionDescriptor**( ) As YFUN_DESCR |
| cs | YFUN_DESCR **get_functionDescriptor**( ) |
| java | String **get_functionDescriptor**( ) |
| py | **get_functionDescriptor**( ) |
| php | function **get_functionDescriptor**( ) |
| es | async **get_functionDescriptor**( ) |

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

**Returns :**
an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

## datalogger→get_functionId()
## datalogger→functionId()

YDataLogger

Returns the hardware identifier of the data logger, without reference to the module.

```
js      function get_functionId( )
cpp     string get_functionId( )
m       -(NSString*) functionId
vb      function get_functionId( ) As String
cs      string get_functionId( )
dnp     string get_functionId( )
java    String get_functionId( )
py      get_functionId( )
php     function get_functionId( )
es      async get_functionId( )
```

For example `relay1`

**Returns :**
a string that identifies the data logger (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

## datalogger→get_hardwareId() YDataLogger
## datalogger→hardwareId()

Returns the unique hardware identifier of the data logger in the form SERIAL.FUNCTIONID.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| dnp | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| py | **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| es | async **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the data logger (for example RELAYLO1-123456.relay1).

**Returns :**
a string that uniquely identifies the data logger (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

## datalogger→get_logicalName()
## datalogger→logicalName()

YDataLogger

Returns the logical name of the data logger.

```
js      function get_logicalName( )
cpp     string get_logicalName( )
m       -(NSString*) logicalName
pas     string get_logicalName( ): string
vb      function get_logicalName( ) As String
cs      string get_logicalName( )
dnp     string get_logicalName( )
java    String get_logicalName( )
uwp     async Task<string> get_logicalName( )
py      get_logicalName( )
php     function get_logicalName( )
es      async get_logicalName( )
cmd     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_module()
## datalogger→module()

YDataLogger

Gets the YModule object for the device on which the function is located.

| | |
|---|---|
| js | function **get_module**( ) |
| cpp | YModule * **get_module**( ) |
| m | -(YModule*) module |
| pas | TYModule **get_module**( ): TYModule |
| vb | function **get_module**( ) As YModule |
| cs | YModule **get_module**( ) |
| dnp | YModuleProxy **get_module**( ) |
| java | YModule **get_module**( ) |
| py | **get_module**( ) |
| php | function **get_module**( ) |
| es | async **get_module**( ) |

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

**Returns :**
an instance of YModule

## datalogger→get_module_async() YDataLogger
## datalogger→module_async()

Gets the YModule object for the device on which the function is located (asynchronous version).

```
js  function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as on-line.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking Firefox JavaScript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous JavasSript calls for more details.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## datalogger→get_recording()
## datalogger→recording()

YDataLogger

Returns the current activation state of the data logger.

| | |
|---|---|
| js | function **get_recording**( ) |
| cpp | Y_RECORDING_enum **get_recording**( ) |
| m | -(Y_RECORDING_enum) recording |
| pas | Integer **get_recording**( ): Integer |
| vb | function **get_recording**( ) As Integer |
| cs | int **get_recording**( ) |
| dnp | int **get_recording**( ) |
| java | int **get_recording**( ) |
| uwp | async Task<int> **get_recording**( ) |
| py | **get_recording**( ) |
| php | function **get_recording**( ) |
| es | async **get_recording**( ) |
| cmd | YDataLogger **target get_recording** |

**Returns :**

a value among `Y_RECORDING_OFF`, `Y_RECORDING_ON` and `Y_RECORDING_PENDING` corresponding to the current activation state of the data logger

On failure, throws an exception or returns `Y_RECORDING_INVALID`.

## datalogger→get_serialNumber()
## datalogger→serialNumber()

YDataLogger

Returns the serial number of the module, as set by the factory.

```
js      function get_serialNumber( )
cpp     string get_serialNumber( )
m       -(NSString*) serialNumber
pas     string get_serialNumber( ): string
vb      function get_serialNumber( ) As String
cs      string get_serialNumber( )
dnp     string get_serialNumber( )
java    String get_serialNumber( )
uwp     async Task<string> get_serialNumber( )
py      get_serialNumber( )
php     function get_serialNumber( )
es      async get_serialNumber( )
cmd     YDataLogger 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 YModule.SERIALNUMBER_INVALID.

## datalogger→get_timeUTC()
## datalogger→timeUTC()

**YDataLogger**

Returns the Unix timestamp for current UTC time, if known.

```
js      function get_timeUTC( )
cpp     s64 get_timeUTC( )
m       -(s64) timeUTC
pas     int64 get_timeUTC( ): int64
vb      function get_timeUTC( ) As Long
cs      long get_timeUTC( )
dnp     long get_timeUTC( )
java    long get_timeUTC( )
uwp     async Task<long> get_timeUTC( )
py      get_timeUTC( )
php     function get_timeUTC( )
es      async get_timeUTC( )
cmd     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→get_usage()
## datalogger→usage()

YDataLogger

Returns the percentage of datalogger memory in use.

```
js      function get_usage( )
cpp     int get_usage( )
m       -(int) usage
pas     LongInt get_usage( ): LongInt
vb      function get_usage( ) As Integer
cs      int get_usage( )
dnp     int get_usage( )
java    int get_usage( )
uwp     async Task<int> get_usage( )
py      get_usage( )
php     function get_usage( )
es      async get_usage( )
cmd     YDataLogger target get_usage
```

**Returns :**
an integer corresponding to the percentage of datalogger memory in use

On failure, throws an exception or returns `Y_USAGE_INVALID`.

## datalogger→get_userData()
## datalogger→userData()

YDataLogger

Returns the value of the userData attribute, as previously stored using method `set_userData`.

| | |
|---|---|
| js | function **get_userData**( ) |
| cpp | void * **get_userData**( ) |
| m | -(id) userData |
| pas | Tobject **get_userData**( ): Tobject |
| vb | function **get_userData**( ) As Object |
| cs | object **get_userData**( ) |
| java | Object **get_userData**( ) |
| py | **get_userData**( ) |
| php | function **get_userData**( ) |
| es | async **get_userData**( ) |

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

**Returns :**
the object stored previously by the caller.

## datalogger→isOnline() YDataLogger

Checks if the data logger is currently reachable, without raising any error.

```
js      function isOnline( )
cpp     bool isOnline( )
m       -(BOOL) isOnline
pas     boolean isOnline( ): boolean
vb      function isOnline( ) As Boolean
cs      bool isOnline( )
dnp     bool isOnline( )
java    boolean isOnline( )
py      isOnline( )
php     function isOnline( )
es      async isOnline( )
```

If there is a cached value for the data logger in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the data logger.

**Returns :**
`true` if the data logger can be reached, and `false` otherwise

## datalogger→isOnline_async() YDataLogger

Checks if the data logger is currently reachable, without raising any error (asynchronous version).

```
js  function isOnline_async( callback, context)
```

If there is a cached value for the data logger in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

**Parameters :**

**callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing : the result is provided to the callback.

## datalogger→isReadOnly() YDataLogger

Test if the function is readOnly.

```
cpp     bool isReadOnly( )
m       -(bool) isReadOnly
pas     boolean isReadOnly( ): boolean
vb      function isReadOnly( ) As Boolean
cs      bool isReadOnly( )
dnp     bool isReadOnly( )
java    boolean isReadOnly( )
uwp     async Task<bool> isReadOnly( )
py      isReadOnly( )
php     function isReadOnly( )
es      async isReadOnly( )
cmd     YDataLogger target isReadOnly
```

Return `true` if the function is write protected or that the function is not available.

**Returns :**
`true` if the function is readOnly or not online.

## datalogger→load() YDataLogger

Preloads the data logger cache with a specified validity duration.

```
js      function load( msValidity)
cpp     YRETCODE load( int msValidity)
m       -(YRETCODE) load : (u64) msValidity
pas     YRETCODE load( msValidity: u64): YRETCODE
vb      function load( ByVal msValidity As Long) As YRETCODE
cs      YRETCODE load( ulong msValidity)
java    int load( long msValidity)
py      load( msValidity)
php     function load( $msValidity)
es      async load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

**Parameters :**

**msValidity** an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

**Returns :**

YAPI_SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

## datalogger→loadAttribute() YDataLogger

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
js      function loadAttribute( attrName)
cpp     string loadAttribute( string attrName)
m       -(NSString*) loadAttribute : (NSString*) attrName
pas     string loadAttribute( attrName: string): string
vb      function loadAttribute( ) As String
cs      string loadAttribute( string attrName)
dnp     string loadAttribute( string attrName)
java    String loadAttribute( String attrName)
uwp     async Task<string> loadAttribute( string attrName)
py      loadAttribute( attrName)
php     function loadAttribute( $attrName)
es      async loadAttribute( attrName)
```

**Parameters :**

**attrName** the name of the requested attribute

**Returns :**

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

## datalogger→load_async() YDataLogger

Preloads the data logger cache with a specified validity duration (asynchronous version).

```
js  function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking the JavaScript virtual machine.

**Parameters :**

| | |
|---|---|
| **msValidity** | an integer corresponding to the validity of the loaded function parameters, in milliseconds |
| **callback** | callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or YAPI_SUCCESS) |
| **context** | caller-specific object that is passed as-is to the callback function |

**Returns :**

nothing : the result is provided to the callback.

## datalogger→muteValueCallbacks() YDataLogger

Disables the propagation of every new advertised value to the parent hub.

```
js      function muteValueCallbacks( )
cpp     int muteValueCallbacks( )
m       -(int) muteValueCallbacks
pas     LongInt muteValueCallbacks( ): LongInt
vb      function muteValueCallbacks( ) As Integer
cs      int muteValueCallbacks( )
dnp     int muteValueCallbacks( )
java    int muteValueCallbacks( )
uwp     async Task<int> muteValueCallbacks( )
py      muteValueCallbacks( )
php     function muteValueCallbacks( )
es      async muteValueCallbacks( )
cmd     YDataLogger target muteValueCallbacks
```

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Returns :**
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

## datalogger→nextDataLogger() YDataLogger

Continues the enumeration of data loggers started using `yFirstDataLogger()`.

| | |
|---|---|
| js | function **nextDataLogger**( ) |
| cpp | YDataLogger * **nextDataLogger**( ) |
| m | -(YDataLogger*) **nextDataLogger** |
| pas | TYDataLogger **nextDataLogger**( ): TYDataLogger |
| vb | function **nextDataLogger**( ) As YDataLogger |
| cs | YDataLogger **nextDataLogger**( ) |
| java | YDataLogger **nextDataLogger**( ) |
| uwp | YDataLogger **nextDataLogger**( ) |
| py | **nextDataLogger**( ) |
| php | function **nextDataLogger**( ) |
| es | **nextDataLogger**( ) |

Caution: You can't make any assumption about the returned data loggers order. If you want to find a specific a data logger, use `DataLogger.findDataLogger()` and a hardwareID or a logical name.

**Returns :**

a pointer to a `YDataLogger` object, corresponding to a data logger currently online, or a `null` pointer if there are no more data loggers to enumerate.

## datalogger→registerValueCallback() YDataLogger

Registers the callback function that is invoked on every change of advertised value.

| | |
|---|---|
| js | function **registerValueCallback**( **callback**) |
| cpp | int **registerValueCallback**( YDataLoggerValueCallback **callback**) |
| m | -(int) **registerValueCallback** : (YDataLoggerValueCallback) **callback** |
| pas | LongInt **registerValueCallback**( **callback**: TYDataLoggerValueCallback): LongInt |
| vb | function **registerValueCallback**( ) As Integer |
| cs | int **registerValueCallback**( ValueCallback **callback**) |
| java | int **registerValueCallback**( UpdateCallback **callback**) |
| uwp | async Task<int> **registerValueCallback**( ValueCallback **callback**) |
| py | **registerValueCallback**( **callback**) |
| php | function **registerValueCallback**( **$callback**) |
| es | async **registerValueCallback**( **callback**) |

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

**Parameters :**

**callback** the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

## datalogger→set_autoStart()
## datalogger→setAutoStart()

YDataLogger

Changes the default activation state of the data logger on power up.

```
js      function set_autoStart( newval)
cpp     int set_autoStart( Y_AUTOSTART_enum newval)
m       -(int) setAutoStart : (Y_AUTOSTART_enum) newval
pas     integer set_autoStart( newval: Integer): integer
vb      function set_autoStart( ByVal newval As Integer) As Integer
cs      int set_autoStart( int newval)
dnp     int set_autoStart( int newval)
java    int set_autoStart( int newval)
uwp     async Task<int> set_autoStart( int newval)
py      set_autoStart( newval)
php     function set_autoStart( $newval)
es      async set_autoStart( newval)
cmd     YDataLogger target set_autoStart newval
```

Do not forget to call the `saveToFlash()` method of the module to save the configuration change. Note: if the device doesn't have any time source at his disposal when starting up, it will wait for ~8 seconds before automatically starting to record with an arbitrary timestamp

**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()
## datalogger→setBeaconDriven()

YDataLogger

Changes the type of synchronisation of the data logger.

```
js    function set_beaconDriven( newval)
cpp   int set_beaconDriven( Y_BEACONDRIVEN_enum newval)
m     -(int) setBeaconDriven : (Y_BEACONDRIVEN_enum) newval
pas   integer set_beaconDriven( newval: Integer): integer
vb    function set_beaconDriven( ByVal newval As Integer) As Integer
cs    int set_beaconDriven( int newval)
dnp   int set_beaconDriven( int newval)
java  int set_beaconDriven( int newval)
uwp   async Task<int> set_beaconDriven( int newval)
py    set_beaconDriven( newval)
php   function set_beaconDriven( $newval)
es    async set_beaconDriven( newval)
cmd   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()
## datalogger→setLogicalName()

YDataLogger

Changes the logical name of the data logger.

| | |
|---|---|
| js | function **set_logicalName**( **newval**) |
| cpp | int **set_logicalName**( string **newval**) |
| m | -(int) setLogicalName : (NSString*) **newval** |
| pas | integer **set_logicalName**( **newval**: string): integer |
| vb | function **set_logicalName**( ByVal **newval** As String) As Integer |
| cs | int **set_logicalName**( string **newval**) |
| dnp | int **set_logicalName**( string **newval**) |
| java | int **set_logicalName**( String **newval**) |
| uwp | async Task<int> **set_logicalName**( string **newval**) |
| py | **set_logicalName**( **newval**) |
| php | function **set_logicalName**( **$newval**) |
| es | async **set_logicalName**( **newval**) |
| cmd | 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()
## datalogger→setRecording()

YDataLogger

Changes the activation state of the data logger to start/stop recording data.

```
js      function set_recording( newval)
cpp     int set_recording( Y_RECORDING_enum newval)
m       -(int) setRecording : (Y_RECORDING_enum) newval
pas     integer set_recording( newval: Integer): integer
vb      function set_recording( ByVal newval As Integer) As Integer
cs      int set_recording( int newval)
dnp     int set_recording( int newval)
java    int set_recording( int newval)
uwp     async Task<int> set_recording( int newval)
py      set_recording( newval)
php     function set_recording( $newval)
es      async set_recording( newval)
cmd     YDataLogger target set_recording newval
```

**Parameters :**

**newval** a value among Y_RECORDING_OFF, Y_RECORDING_ON and Y_RECORDING_PENDING corresponding 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()
## datalogger→setTimeUTC()

YDataLogger

Changes the current UTC time reference used for recorded data.

```
js      function set_timeUTC( newval)
cpp     int set_timeUTC( s64 newval)
m       -(int) setTimeUTC : (s64) newval
pas     integer set_timeUTC( newval: int64): integer
vb      function set_timeUTC( ByVal newval As Long) As Integer
cs      int set_timeUTC( long newval)
dnp     int set_timeUTC( long newval)
java    int set_timeUTC( long newval)
uwp     async Task<int> set_timeUTC( long newval)
py      set_timeUTC( newval)
php     function set_timeUTC( $newval)
es      async set_timeUTC( newval)
cmd     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.

## datalogger→set_userData()
## datalogger→setUserData()

YDataLogger

Stores a user context provided as argument in the userData attribute of the function.

```
js      function set_userData( data)
cpp     void set_userData( void * data)
m       -(void) setUserData : (id) data
pas     set_userData( data: Tobject)
vb      procedure set_userData( ByVal data As Object)
cs      void set_userData( object data)
java    void set_userData( Object data)
py      set_userData( data)
php     function set_userData( $data)
es      async set_userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

**Parameters :**

**data** any kind of object to be stored

## datalogger→unmuteValueCallbacks() YDataLogger

Re-enables the propagation of every new advertised value to the parent hub.

| | |
|---|---|
| js | function **unmuteValueCallbacks**( ) |
| cpp | int **unmuteValueCallbacks**( ) |
| m | -(int) **unmuteValueCallbacks** |
| pas | LongInt **unmuteValueCallbacks**( ): LongInt |
| vb | function **unmuteValueCallbacks**( ) As Integer |
| cs | int **unmuteValueCallbacks**( ) |
| dnp | int **unmuteValueCallbacks**( ) |
| java | int **unmuteValueCallbacks**( ) |
| uwp | async Task<int> **unmuteValueCallbacks**( ) |
| py | **unmuteValueCallbacks**( ) |
| php | function **unmuteValueCallbacks**( ) |
| es | async **unmuteValueCallbacks**( ) |
| cmd | YDataLogger **target unmuteValueCallbacks** |

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

**Returns :**
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

## datalogger→wait_async() YDataLogger

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js  function wait_async( callback, context)
es  wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

**Parameters :**

**callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

**context** caller-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 22.9. Class YDataSet

Recorded data sequence, as returned by `sensor.get_recordedData()`

`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 `sensor.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 relatively 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_module.js'></script> |
| cpp | #include "yocto_module.h" |
| m | #import "yocto_module.h" |
| pas | uses yocto_module; |
| vb | yocto_module.vb |
| cs | yocto_module.cs |
| dnp | import YoctoProxyAPI.YDataSetProxy |
| java | import com.yoctopuce.YoctoAPI.YDataSet; |
| uwp | import com.yoctopuce.YoctoAPI.YDataSet; |
| py | from yocto_module import * |
| php | require_once('yocto_module.php'); |
| es | in HTML: <script src="../../lib/yocto_module.js"></script><br>in node.js: require('yoctolib-es2017/yocto_module.js'); |

## Global functions

**YDataSet.Init(sensorName, startTime, endTime)**

Retrieves a `YDataSet` object holding historical data for a sensor given by its name or hardware identifier, for a specified time interval.

## `YDataSet` properties

**dataset→FunctionId** *[read-only]*

Hardware identifier of the function that performed the measure, without reference to the module.

**dataset→HardwareId** *[read-only]*

Unique hardware identifier of the function who performed the measures, in the form `SERIAL.FUNCTIONID`.

**dataset→MeasuresRecordCount** *[read-only]*

Number of measurements currently loaded for this data set.

**dataset→PreviewRecordCount** *[read-only]*

**dataset→Progress** *[read-only]*

Progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

**dataset→SummaryAvg** *[read-only]*

Average value observed during the time interval covered by this data set.

**dataset→SummaryEndTime** *[read-only]*

End time of the last measure in the data set, relative to the Jan 1, 1970 UTC (Unix timestamp).

**dataset→SummaryMax** *[read-only]*

Largest value observed during the time interval covered by this data set.

**dataset→SummaryMin** *[read-only]*

Smallest value observed during the time interval covered by this data set.

**dataset→SummaryStartTime** *[read-only]*

Start time of the first measure in the data set, relative to the Jan 1, 1970 UTC (Unix timestamp).

**dataset→Unit** *[read-only]*

Measuring unit for the measured value.

### `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_measuresAt**(**measure**)

Returns the detailed set of measures for the time interval corresponding to a given condensed measures previously returned by `get_preview()`.

**dataset→get_measuresAvgAt**(**index**)

Returns the average value observed during the time interval covered by the specified entry in the preview.

**dataset→get_measuresEndTimeAt**(**index**)

Returns the end time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

**dataset→get_measuresMaxAt**(**index**)

Returns the largest value observed during the time interval covered by the specified entry in the preview.

**dataset→get_measuresMinAt**(**index**)

Returns the smallest value observed during the time interval covered by the specified entry in the preview.

**dataset→get_measuresRecordCount**()

Returns the number of measurements currently loaded for this data set.

**dataset→get_measuresStartTimeAt**(**index**)

Returns the start time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

**dataset→get_preview**()

Returns a condensed version of the measures that can retrieved in this `YDataSet`, as a list of `YMeasure` objects.

**dataset→get_previewAvgAt**(**index**)

Returns the average value observed during the time interval covered by the specified entry in the preview.

**dataset→get_previewEndTimeAt**(**index**)

Returns the end time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

**dataset→get_previewMaxAt**(**index**)

Returns the largest value observed during the time interval covered by the specified entry in the preview.

**dataset→get_previewMinAt**(**index**)

Returns the smallest value observed during the time interval covered by the specified entry in the preview.

**dataset→get_previewRecordCount**()

Returns the number of entries in the preview summarizing this data set

**dataset→get_previewStartTimeAt**(**index**)

Returns the start time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

**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 `YDataSet`.

**dataset→get_summaryAvg**()

Returns the average value observed during the time interval covered by this data set.

**dataset→get_summaryEndTime**()

Returns the end time of the last measure in the data set, relative to the Jan 1, 1970 UTC (Unix timestamp).

**dataset→get_summaryMax**()

Returns the largest value observed during the time interval covered by this data set.

**dataset→get_summaryMin**()

Returns the smallest value observed during the time interval covered by this data set.

**dataset→get_summaryStartTime**()

Returns the start time of the first measure in the data set, relative to the Jan 1, 1970 UTC (Unix timestamp).

**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.

## YDataSet.Init() YDataSet
## YDataSet.Init()

Retrieves a YDataSet object holding historical data for a sensor given by its name or hardware identifier, for a specified time interval.

```
dnp  static YDataSetProxy Init( string sensorName,
                                double startTime,
                                double endTime)
```

The measures will be retrieved from the data logger, which must have been turned on at the desired time. Methods of the YDataSet class makes it possible to get an overview of the recorded data, and to load progressively a large set of measures from the data logger.

**Parameters :**

**sensorName** logical name or hardware identifier of the sensor for which data logger records are requested.

**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 measure, 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 measure, 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.

## dataset→FunctionId — YDataSet

Hardware identifier of the function that performed the measure, without reference to the module.

dnp string **FunctionId**

For example `temperature1`.

## dataset→HardwareId YDataSet

Unique hardware identifier of the function who performed the measures, in the form SERIAL.FUNCTIONID.

dnp string **HardwareId**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example THRMCPL1-123456.temperature1)

## dataset→MeasuresRecordCount — YDataSet

Number of measurements currently loaded for this data set.

dnp int **MeasuresRecordCount**

The total number of record is only known when the data set is fully loaded, i.e. when `loadMore()` has been invoked until the progresss indicator returns 100.

## dataset→PreviewRecordCount — YDataSet

dnp int **PreviewRecordCount**

Number of entries in the preview summarizing this data set

## dataset→Progress YDataSet

Progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

dnp int **Progress**

When the object is instantiated by `get_dataSet`, the progress is zero. Each time `loadMore()` is invoked, the progress is updated, to reach the value 100 only once all measures have been loaded.

## dataset→SummaryAvg — YDataSet

Average value observed during the time interval covered by this data set.

dnp double **SummaryAvg**

## dataset→SummaryEndTime YDataSet

End time of the last measure in the data set, relative to the Jan 1, 1970 UTC (Unix timestamp).

dnp double **SummaryEndTime**

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

## dataset→SummaryMax — YDataSet

Largest value observed during the time interval covered by this data set.

dnp double **SummaryMax**

## dataset→SummaryMin — YDataSet

Smallest value observed during the time interval covered by this data set.

dnp double **SummaryMin**

## dataset→SummaryStartTime — YDataSet

Start time of the first measure in the data set, relative to the Jan 1, 1970 UTC (Unix timestamp).

dnp double **SummaryStartTime**

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

## dataset→Unit — YDataSet

Measuring unit for the measured value.

dnp string **Unit**

## dataset→get_endTimeUTC() YDataSet
## dataset→endTimeUTC()

Returns the end time of the dataset, relative to the Jan 1, 1970.

| | |
|---|---|
| js | function **get_endTimeUTC**( ) |
| cpp | s64 **get_endTimeUTC**( ) |
| m | -(s64) endTimeUTC |
| pas | int64 **get_endTimeUTC**( ): int64 |
| vb | function **get_endTimeUTC**( ) As Long |
| cs | long **get_endTimeUTC**( ) |
| dnp | long **get_endTimeUTC**( ) |
| java | long **get_endTimeUTC**( ) |
| uwp | async Task<long> **get_endTimeUTC**( ) |
| py | **get_endTimeUTC**( ) |
| php | function **get_endTimeUTC**( ) |
| es | async **get_endTimeUTC**( ) |

When the `YDataSet` object is created, the end time is the value passed in parameter to the `get_dataSet()` function. After the very first call to `loadMore()`, the end time is updated to reflect the timestamp of the last measure actually found in the dataLogger within the specified range.

**DEPRECATED**: This method has been replaced by `get_summary()` which contain more precise informations.

**Returns :**

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the end of this data set (i.e. Unix time representation of the absolute time).

## dataset→get_functionId()
## dataset→functionId()

**YDataSet**

Returns the hardware identifier of the function that performed the measure, without reference to the module.

| | |
|---|---|
| js | function **get_functionId**( ) |
| cpp | string **get_functionId**( ) |
| m | -(NSString*) functionId |
| pas | string **get_functionId**( ): string |
| vb | function **get_functionId**( ) As String |
| cs | string **get_functionId**( ) |
| dnp | string **get_functionId**( ) |
| java | String **get_functionId**( ) |
| uwp | async Task<string> **get_functionId**( ) |
| py | **get_functionId**( ) |
| php | function **get_functionId**( ) |
| es | async **get_functionId**( ) |

For example `temperature1`.

**Returns :**
a string that identifies the function (ex: `temperature1`)

## dataset→get_hardwareId()
## dataset→hardwareId()

YDataSet

Returns the unique hardware identifier of the function who performed the measures, in the form SERIAL.FUNCTIONID.

| | |
|---|---|
| js | function **get_hardwareId**( ) |
| cpp | string **get_hardwareId**( ) |
| m | -(NSString*) hardwareId |
| pas | string **get_hardwareId**( ): string |
| vb | function **get_hardwareId**( ) As String |
| cs | string **get_hardwareId**( ) |
| dnp | string **get_hardwareId**( ) |
| java | String **get_hardwareId**( ) |
| uwp | async Task<string> **get_hardwareId**( ) |
| py | **get_hardwareId**( ) |
| php | function **get_hardwareId**( ) |
| es | async **get_hardwareId**( ) |

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example THRMCPL1-123456.temperature1)

**Returns :**
a string that uniquely identifies the function (ex: THRMCPL1-123456.temperature1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

## dataset→get_measures()
## dataset→measures()

**YDataSet**

Returns all measured values currently available for this DataSet, as a list of `YMeasure` objects.

| | |
|---|---|
| js | function **get_measures**( ) |
| cpp | vector<YMeasure> **get_measures**( ) |
| m | -(NSMutableArray*) measures |
| pas | TYMeasureArray **get_measures**( ): TYMeasureArray |
| vb | function **get_measures**( ) As List |
| cs | List<YMeasure> **get_measures**( ) |
| dnp | YMeasure[] **get_measures**( ) |
| java | ArrayList<YMeasure> **get_measures**( ) |
| uwp | async Task<List<YMeasure>> **get_measures**( ) |
| py | **get_measures**( ) |
| php | function **get_measures**( ) |
| es | async **get_measures**( ) |

Each item includes: - the start of the measure time interval - the end of the measure time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

Before calling this method, you should call `loadMore()` to load data from the device. You may have to call loadMore() several time until all rows are loaded, but you can start looking at available data rows before the load is complete.

The oldest measures are always loaded first, and the most recent measures will be loaded last. As a result, timestamps are normally sorted in ascending order within the measure table, unless there was an unexpected adjustment of the datalogger UTC clock.

**Returns :**
a table of records, where each record depicts the measured value for a given time interval

On failure, throws an exception or returns an empty array.

## dataset→get_measuresAt() YDataSet
## dataset→measuresAt()

Returns the detailed set of measures for the time interval corresponding to a given condensed measures previously returned by `get_preview()`.

| | |
|---|---|
| js | function **get_measuresAt**( **measure**) |
| cpp | vector<YMeasure> **get_measuresAt**( YMeasure **measure**) |
| m | -(NSMutableArray*) measuresAt : (YMeasure*) **measure** |
| pas | TYMeasureArray **get_measuresAt**( **measure**: TYMeasure): TYMeasureArray |
| vb | function **get_measuresAt**( ) As List |
| cs | List<YMeasure> **get_measuresAt**( YMeasure **measure**) |
| dnp | YMeasure[] **get_measuresAt**( YMeasure **measure**) |
| java | ArrayList<YMeasure> **get_measuresAt**( YMeasure **measure**) |
| uwp | async Task<List<YMeasure>> **get_measuresAt**( YMeasure **measure**) |
| py | **get_measuresAt**( **measure**) |
| php | function **get_measuresAt**( **$measure**) |
| es | async **get_measuresAt**( **measure**) |

The result is provided as a list of `YMeasure` objects.

**Parameters :**

**measure** condensed measure from the list previously returned by `get_preview()`.

**Returns :**

a table of records, where each record depicts the measured values during a time interval

On failure, throws an exception or returns an empty array.

## dataset→get_measuresAvgAt() YDataSet
## dataset→measuresAvgAt()

Returns the average value observed during the time interval covered by the specified entry in the preview.

dnp double **get_measuresAvgAt**( int **index**)

**Parameters :**

**index** an integer index in the range [0...MeasuresRecordCount-1].

**Returns :**

a floating-point number corresponding to the average value observed.

## dataset→get_measuresEndTimeAt() YDataSet
## dataset→measuresEndTimeAt()

Returns the end time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

dnp double **get_measuresEndTimeAt**( int **index**)

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

**Parameters :**

**index** an integer index in the range [0...MeasuresRecordCount-1].

**Returns :**

a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

## dataset→get_measuresMaxAt() YDataSet
## dataset→measuresMaxAt()

Returns the largest value observed during the time interval covered by the specified entry in the preview.

dnp double **get_measuresMaxAt**( int **index**)

**Parameters :**

**index** an integer index in the range [0...MeasuresRecordCount-1].

**Returns :**

a floating-point number corresponding to the largest value observed.

## dataset→get_measuresMinAt() YDataSet
## dataset→measuresMinAt()

Returns the smallest value observed during the time interval covered by the specified entry in the preview.

dnp double **get_measuresMinAt**( int **index**)

**Parameters :**

**index** an integer index in the range [0...MeasuresRecordCount-1].

**Returns :**

a floating-point number corresponding to the smallest value observed.

## dataset→get_measuresRecordCount() YDataSet
## dataset→measuresRecordCount()

Returns the number of measurements currently loaded for this data set.

dnp int **get_measuresRecordCount**( )

The total number of record is only known when the data set is fully loaded, i.e. when `loadMore()` has been invoked until the progresss indicator returns 100.

**Returns :**
an integer number corresponding to the number of entries loaded.

## dataset→get_measuresStartTimeAt() YDataSet
## dataset→measuresStartTimeAt()

Returns the start time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

dnp double **get_measuresStartTimeAt**( int **index**)

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

**Parameters :**

**index** an integer index in the range [0...MeasuresRecordCount-1].

**Returns :**

a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

## dataset→get_preview()
## dataset→preview()

**YDataSet**

Returns a condensed version of the measures that can retrieved in this `YDataSet`, as a list of `YMeasure` objects.

| | |
|---|---|
| js | function **get_preview**( ) |
| cpp | vector<YMeasure> **get_preview**( ) |
| m | -(NSMutableArray*) preview |
| pas | TYMeasureArray **get_preview**( ): TYMeasureArray |
| vb | function **get_preview**( ) As List |
| cs | List<YMeasure> **get_preview**( ) |
| dnp | YMeasure[] **get_preview**( ) |
| java | ArrayList<YMeasure> **get_preview**( ) |
| uwp | async Task<List<YMeasure>> **get_preview**( ) |
| py | **get_preview**( ) |
| php | function **get_preview**( ) |
| es | async **get_preview**( ) |

Each item includes: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This preview is available as soon as `loadMore()` has been called for the first time.

**Returns :**
a table of records, where each record depicts the measured values during a time interval

On failure, throws an exception or returns an empty array.

## dataset→get_previewAvgAt() YDataSet
## dataset→previewAvgAt()

Returns the average value observed during the time interval covered by the specified entry in the preview.

dnp double **get_previewAvgAt**( int **index**)

**Parameters :**

**index** an integer index in the range [0...PreviewRecordCount-1].

**Returns :**

a floating-point number corresponding to the average value observed.

## dataset→get_previewEndTimeAt() YDataSet
## dataset→previewEndTimeAt()

Returns the end time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

dnp double **get_previewEndTimeAt**( int **index**)

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

**Parameters :**

**index** an integer index in the range [0...PreviewRecordCount-1].

**Returns :**

a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

## dataset→get_previewMaxAt() YDataSet
## dataset→previewMaxAt()

Returns the largest value observed during the time interval covered by the specified entry in the preview.

dnp double **get_previewMaxAt**( int **index**)

**Parameters :**

**index** an integer index in the range [0...PreviewRecordCount-1].

**Returns :**

a floating-point number corresponding to the largest value observed.

## dataset→get_previewMinAt() YDataSet
## dataset→previewMinAt()

Returns the smallest value observed during the time interval covered by the specified entry in the preview.

dnp double **get_previewMinAt**( int **index**)

**Parameters :**

**index** an integer index in the range [0...PreviewRecordCount-1].

**Returns :**

a floating-point number corresponding to the smallest value observed.

## dataset→get_previewRecordCount()
## dataset→previewRecordCount()

YDataSet

Returns the number of entries in the preview summarizing this data set

dnp int **get_previewRecordCount**( )

**Returns :**
an integer number corresponding to the number of entries.

## dataset→get_previewStartTimeAt() YDataSet
## dataset→previewStartTimeAt()

Returns the start time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

dnp double **get_previewStartTimeAt**( int **index**)

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

**Parameters :**

**index** an integer index in the range [0...PreviewRecordCount-1].

**Returns :**

a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

## dataset→get_progress() YDataSet
## dataset→progress()

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

| | |
|---|---|
| js | function **get_progress**( ) |
| cpp | int **get_progress**( ) |
| m | -(int) progress |
| pas | LongInt **get_progress**( ): LongInt |
| vb | function **get_progress**( ) As Integer |
| cs | int **get_progress**( ) |
| dnp | int **get_progress**( ) |
| java | int **get_progress**( ) |
| uwp | async Task<int> **get_progress**( ) |
| py | **get_progress**( ) |
| php | function **get_progress**( ) |
| es | async **get_progress**( ) |

When the object is instantiated by `get_dataSet`, the progress is zero. Each time `loadMore()` is invoked, the progress is updated, to reach the value 100 only once all measures have been loaded.

**Returns :**
an integer in the range 0 to 100 (percentage of completion).

## dataset→get_startTimeUTC() YDataSet
## dataset→startTimeUTC()

Returns the start time of the dataset, relative to the Jan 1, 1970.

| | |
|---|---|
| js | function **get_startTimeUTC**( ) |
| cpp | s64 **get_startTimeUTC**( ) |
| m | -(s64) startTimeUTC |
| pas | int64 **get_startTimeUTC**( ): int64 |
| vb | function **get_startTimeUTC**( ) As Long |
| cs | long **get_startTimeUTC**( ) |
| dnp | long **get_startTimeUTC**( ) |
| java | long **get_startTimeUTC**( ) |
| uwp | async Task<long> **get_startTimeUTC**( ) |
| py | **get_startTimeUTC**( ) |
| php | function **get_startTimeUTC**( ) |
| es | async **get_startTimeUTC**( ) |

When the `YDataSet` object is created, the start time is the value passed in parameter to the `get_dataSet()` function. After the very first call to `loadMore()`, the start time is updated to reflect the timestamp of the first measure actually found in the dataLogger within the specified range.

**DEPRECATED**: This method has been replaced by `get_summary()` which contain more precise informations.

**Returns :**
an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the beginning of this data set (i.e. Unix time representation of the absolute time).

## dataset→get_summary()
## dataset→summary()

YDataSet

Returns an `YMeasure` object which summarizes the whole `YDataSet`.

| | |
|---|---|
| js | function **get_summary**( ) |
| cpp | YMeasure **get_summary**( ) |
| m | -(YMeasure*) summary |
| pas | TYMeasure **get_summary**( ): TYMeasure |
| vb | function **get_summary**( ) As YMeasure |
| cs | YMeasure **get_summary**( ) |
| dnp | YMeasure **get_summary**( ) |
| java | YMeasure **get_summary**( ) |
| uwp | async Task<YMeasure> **get_summary**( ) |
| py | **get_summary**( ) |
| php | function **get_summary**( ) |
| es | async **get_summary**( ) |

In includes the following information: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This summary is available as soon as `loadMore()` has been called for the first time.

**Returns :**
an `YMeasure` object

## dataset→get_summaryAvg()
## dataset→summaryAvg()

YDataSet

Returns the average value observed during the time interval covered by this data set.

dnp double **get_summaryAvg**( )

**Returns :**
a floating-point number corresponding to the average value observed.

## dataset→get_summaryEndTime()
## dataset→summaryEndTime()

YDataSet

Returns the end time of the last measure in the data set, relative to the Jan 1, 1970 UTC (Unix timestamp).

dnp double **get_summaryEndTime**( )

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

**Returns :**
a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

## dataset→get_summaryMax()
## dataset→summaryMax()

YDataSet

Returns the largest value observed during the time interval covered by this data set.

dnp double **get_summaryMax**( )

**Returns :**
a floating-point number corresponding to the largest value observed.

## dataset→get_summaryMin()
## dataset→summaryMin()

YDataSet

Returns the smallest value observed during the time interval covered by this data set.

dnp double **get_summaryMin**( )

**Returns :**
a floating-point number corresponding to the smallest value observed.

## dataset→get_summaryStartTime() YDataSet
## dataset→summaryStartTime()

Returns the start time of the first measure in the data set, relative to the Jan 1, 1970 UTC (Unix timestamp).

dnp double **get_summaryStartTime**( )

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

**Returns :**
a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

## dataset→get_unit()
## dataset→unit()

YDataSet

Returns the measuring unit for the measured value.

```
js      function get_unit( )
cpp     string get_unit( )
m       -(NSString*) unit
pas     string get_unit( ): string
vb      function get_unit( ) As String
cs      string get_unit( )
dnp     string get_unit( )
java    String get_unit( )
uwp     async Task<string> get_unit( )
py      get_unit( )
php     function get_unit( )
es      async get_unit( )
```

**Returns :**
a string that represents a physical unit.

On failure, throws an exception or returns `Y_UNIT_INVALID`.

## dataset→loadMore() YDataSet

Loads the the next block of measures from the dataLogger, and updates the progress indicator.

```
js      function loadMore( )
cpp     int loadMore( )
m       -(int) loadMore
pas     LongInt loadMore( ): LongInt
vb      function loadMore( ) As Integer
cs      int loadMore( )
dnp     int loadMore( )
java    int loadMore( )
uwp     async Task<int> loadMore( )
py      loadMore( )
php     function loadMore( )
es      async loadMore( )
```

**Returns :**

an integer in the range 0 to 100 (percentage of completion), or a negative error code in case of failure.

On failure, throws an exception or returns a negative error code.

## dataset→loadMore_async() YDataSet

Loads the the next block of measures from the dataLogger asynchronously.

```
js  function loadMore_async( callback, context)
```

**Parameters :**

**callback** callback function that is invoked when the w The callback function receives three arguments: - the user-specific context object - the `YDataSet` object whose `loadMore_async` was invoked - the load result: either the progress indicator (0...100), or a negative error code in case of failure.

**context** user-specific object that is passed as-is to the callback function

**Returns :**

nothing.

# 22.10. Class YMeasure

Measured value, returned in particular by the methods of the `YDataSet` class.

`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, but also for sensors periodic timed reports (see `sensor.registerTimedReportCallback`).

In order to use the functions described here, you should include:

| | |
|---|---|
| `js` | <script type='text/javascript' src='yocto_module.js'></script> |
| `cpp` | #include "yocto_module.h" |
| `m` | #import "yocto_module.h" |
| `pas` | uses yocto_module; |
| `vb` | yocto_module.vb |
| `cs` | yocto_module.cs |
| `java` | import com.yoctopuce.YoctoAPI.YMeasure; |
| `uwp` | import com.yoctopuce.YoctoAPI.YMeasure; |
| `py` | from yocto_module import * |
| `php` | require_once('yocto_module.php'); |
| `es` | in HTML: <script src="../../lib/yocto_module.js"></script><br>in node.js: require('yoctolib-es2017/yocto_module.js'); |

| `YMeasure` methods |
|---|
| **measure→get_averageValue**()<br>Returns the average value observed during the time interval covered by this measure. |
| **measure→get_endTimeUTC**()<br>Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp). |
| **measure→get_maxValue**()<br>Returns the largest value observed during the time interval covered by this measure. |
| **measure→get_minValue**()<br>Returns the smallest value observed during the time interval covered by this measure. |
| **measure→get_startTimeUTC**()<br>Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp). |

## measure→get_averageValue()
## measure→averageValue()

YMeasure

Returns the average value observed during the time interval covered by this measure.

| | |
|---|---|
| js | function **get_averageValue**( ) |
| cpp | double **get_averageValue**( ) |
| m | -(double) averageValue |
| pas | double **get_averageValue**( ): double |
| vb | function **get_averageValue**( ) As Double |
| cs | double **get_averageValue**( ) |
| java | double **get_averageValue**( ) |
| uwp | double **get_averageValue**( ) |
| py | **get_averageValue**( ) |
| php | function **get_averageValue**( ) |
| es | **get_averageValue**( ) |

**Returns :**
a floating-point number corresponding to the average value observed.

## measure→get_endTimeUTC()
## measure→endTimeUTC()

YMeasure

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

| | |
|---|---|
| js | function **get_endTimeUTC**( ) |
| cpp | double **get_endTimeUTC**( ) |
| m | -(double) endTimeUTC |
| pas | double **get_endTimeUTC**( ): double |
| vb | function **get_endTimeUTC**( ) As Double |
| cs | double **get_endTimeUTC**( ) |
| java | double **get_endTimeUTC**( ) |
| uwp | double **get_endTimeUTC**( ) |
| py | **get_endTimeUTC**( ) |
| php | function **get_endTimeUTC**( ) |
| es | **get_endTimeUTC**( ) |

When the recording rate is higher than 1 sample per second, the timestamp may have a fractional part.

**Returns :**

a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the end of this measure.

## measure→get_maxValue()
## measure→maxValue()

YMeasure

Returns the largest value observed during the time interval covered by this measure.

| | |
|---|---|
| js | function **get_maxValue**( ) |
| cpp | double **get_maxValue**( ) |
| m | -(double) maxValue |
| pas | double **get_maxValue**( ): double |
| vb | function **get_maxValue**( ) As Double |
| cs | double **get_maxValue**( ) |
| java | double **get_maxValue**( ) |
| uwp | double **get_maxValue**( ) |
| py | **get_maxValue**( ) |
| php | function **get_maxValue**( ) |
| es | **get_maxValue**( ) |

**Returns :**
a floating-point number corresponding to the largest value observed.

## measure→get_minValue()
## measure→minValue()

YMeasure

Returns the smallest value observed during the time interval covered by this measure.

| | |
|---|---|
| js | function **get_minValue**( ) |
| cpp | double **get_minValue**( ) |
| m | -(double) minValue |
| pas | double **get_minValue**( ): double |
| vb | function **get_minValue**( ) As Double |
| cs | double **get_minValue**( ) |
| java | double **get_minValue**( ) |
| uwp | double **get_minValue**( ) |
| py | **get_minValue**( ) |
| php | function **get_minValue**( ) |
| es | **get_minValue**( ) |

**Returns :**
a floating-point number corresponding to the smallest value observed.

## measure→get_startTimeUTC() YMeasure
## measure→startTimeUTC()

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

| | |
|---|---|
| js | function **get_startTimeUTC**( ) |
| cpp | double **get_startTimeUTC**( ) |
| m | -(double) startTimeUTC |
| pas | double **get_startTimeUTC**( ): double |
| vb | function **get_startTimeUTC**( ) As Double |
| cs | double **get_startTimeUTC**( ) |
| java | double **get_startTimeUTC**( ) |
| uwp | double **get_startTimeUTC**( ) |
| py | **get_startTimeUTC**( ) |
| php | function **get_startTimeUTC**( ) |
| es | **get_startTimeUTC**( ) |

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

**Returns :**

a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

# 23. Troubleshooting

## 23.1. Where to start?

If it is the first time that you use a Yoctopuce module and you do not really know where to start, have a look at the Yoctopuce blog. There is a section dedicated to beginners [1].

## 23.2. Programming examples don't seem to work

Most of Yoctopuce API programming examples are command line programs and require some parameters to work properly. You have to start them from your operationg system command prompt, or configure your IDE to run them with the proper parameters. [2].

## 23.3. Linux and USB

To work correctly under Linux, the the library needs to have write access to all the Yoctopuce USB peripherals. However, by default under Linux, USB privileges of the non-root users are limited to read access. To avoid having to run the *VirtualHub* as root, you need to create a new *udev* rule to authorize one or several users to have write access to the Yoctopuce peripherals.

To add a new *udev* rule to your installation, you must add a file with a name following the "`##-arbitraryName.rules`" format, in the "`/etc/udev/rules.d`" directory. When the system is starting, *udev* reads all the files with a "`.rules`" extension in this directory, respecting the alphabetical order (for example, the "`51-custom.rules`" file is interpreted AFTER the "`50-udev-default.rules`" file).

The "`50-udev-default`" file contains the system default *udev* rules. To modify the default behavior, you therefore need to create a file with a name that starts with a number larger than 50, that will override the system default rules. Note that to add a rule, you need a root access on the system.

In the `udev_conf` directory of the *VirtualHub* for Linux[3] archive, there are two rule examples which you can use as a basis.

[1] see: http://www.yoctopuce.com/EN/blog_by_categories/for-the-beginners
[2] see: http://www.yoctopuce.com/EN/article/about-programming-examples
[3] http://www.yoctopuce.com/FR/virtualhub.php

### Example 1: 51-yoctopuce.rules

This rule provides all the users with read and write access to the Yoctopuce USB peripherals. Access rights for all other peripherals are not modified. If this scenario suits you, you only need to copy the "51-yoctopuce_all.rules" file into the "/etc/udev/rules.d" directory and to restart your system.

```
# udev rules to allow write access to all users
# for Yoctopuce USB devices
SUBSYSTEM=="usb", ATTR{idVendor}=="24e0", MODE="0666"
```

### Example 2: 51-yoctopuce_group.rules

This rule authorizes the "yoctogroup" group to have read and write access to Yoctopuce USB peripherals. Access rights for all other peripherals are not modified. If this scenario suits you, you only need to copy the "51-yoctopuce_group.rules" file into the "/etc/udev/rules.d" directory and restart your system.

```
# udev rules to allow write access to all users of "yoctogroup"
# for Yoctopuce USB devices
SUBSYSTEM=="usb", ATTR{idVendor}=="24e0", MODE="0664", GROUP="yoctogroup"
```

## 23.4. ARM Platforms: HF and EL

There are two main flavors of executable on ARM: HF (Hard Float) binaries, and EL (EABI Little Endian) binaries. These two families are not compatible at all. The compatibility of a given ARM platform with of one of these two families depends on the hardware and on the OS build. ArmHL and ArmEL compatibility problems are quite difficult to detect. Most of the time, the OS itself is unable to make a difference between an HF and an EL executable and will return meaningless messages when you try to use the wrong type of binary.

All pre-compiled Yoctopuce binaries are provided in both formats, as two separate ArmHF et ArmEL executables. If you do not know what family your ARM platform belongs to, just try one executable from each family.

## 23.5. Powered module but invisible for the OS

If your Yocto-GPS is connected by USB, if its blue led is on, but if the operating system cannot see the module, check that you are using a true USB cable with data wires, and not a charging cable. Charging cables have only power wires.

## 23.6. Another process named xxx is already using yAPI

If when initializing the Yoctopuce API, you obtain the "*Another process named xxx is already using yAPI*" error message, it means that another application is already using Yoctopuce USB modules. On a single machine only one process can access Yoctopuce modules by USB at a time. You can easily work around this limitation by using a VirtualHub and the network mode [4].

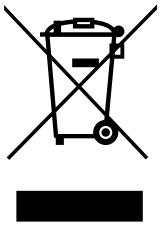
## 23.7. Disconnections, erratic behavior

If you Yocto-GPS behaves erratically and/or disconnects itself from the USB bus without apparent reason, check that it is correctly powered. Avoid cables with a length above 2 meters. If needed, insert a powered USB hub [5] [6].

[4] see: http://www.yoctopuce.com/EN/article/error-message-another-process-is-already-using-yapi
[5] see: http://www.yoctopuce.com/EN/article/usb-cables-size-matters
[6] see: http://www.yoctopuce.com/EN/article/how-many-usb-devices-can-you-connect

## 23.8. Damaged device

Yoctopuce strives to reduce the production of electronic waste. If you believe that your Yocto-GPS is not working anymore, start by contacting Yoctopuce support by e-mail to diagnose the failure. Even if you know that the device was damaged by mistake, Yoctopuce engineers might be able to repair it, and thus avoid creating electronic waste.



**Waste Electrical and Electronic Equipment (WEEE)** If you really want to get rid of your Yocto-GPS, do not throw it away in a trash bin but bring it to your local WEEE recycling point. In this way, it will be disposed properly by a specialized WEEE recycling center.

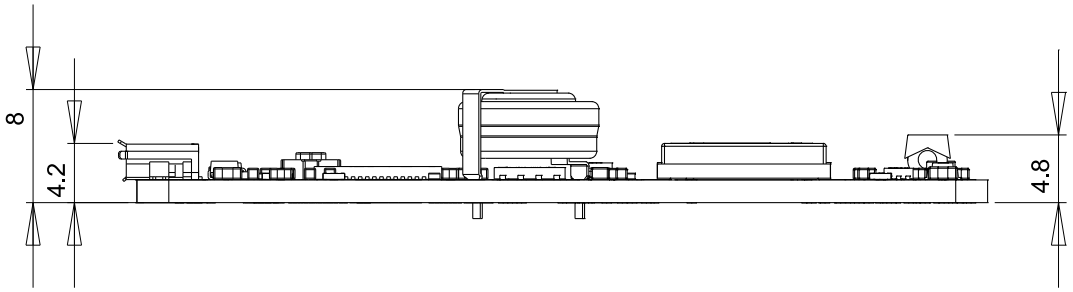
# 24. Characteristics

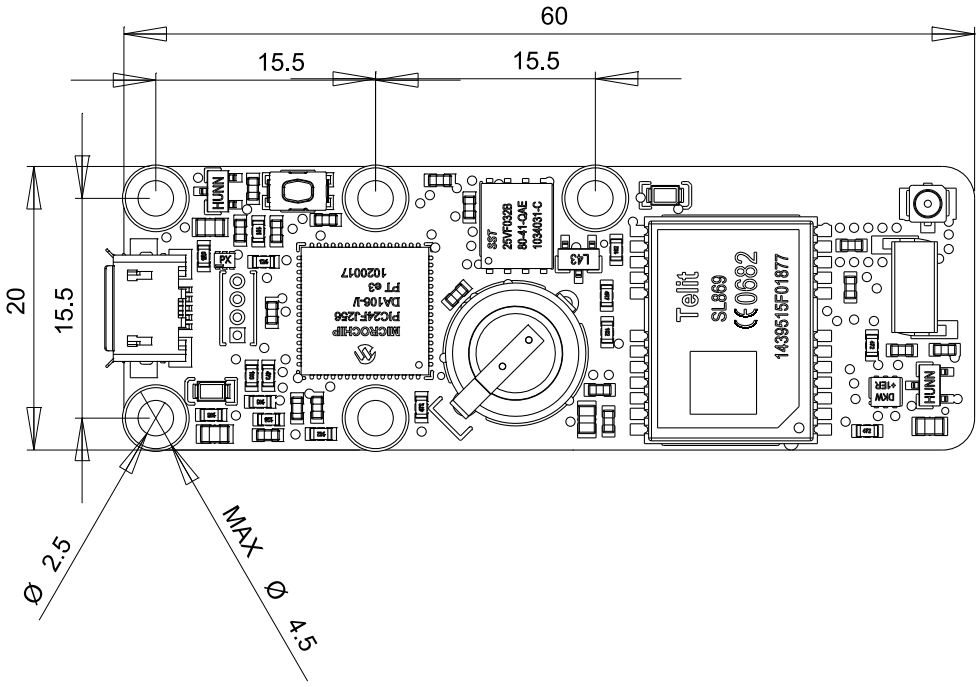
You can find below a summary of the main technical characteristics of your Yocto-GPS module.

| | |
|---|---|
| Product ID | YGNSSMK1 |
| Hardware release† | Rev. B |
| USB connector | micro-B |
| Width | 20 mm |
| Length | 60 mm |
| Weight | 21 g |
| Channels | 32 |
| Chipset | Telit SL869 V3 |
| Refresh rate | 10 Hz |
| Accuracy | 1.5 m |
| Resolution | 10 cm |
| Protection class, according to IEC 61140 | class III |
| Normal operating temperature | 5...40 °C |
| Extended operating temperature‡ | -25...85 °C |
| RoHS compliance | RoHS III (2011/65/UE+2015/863) |
| USB Vendor ID | 0x24E0 |
| USB Device ID | 0x0053 |
| Suggested enclosure | YoctoBox-Long-Thick-Black |
| Harmonized tariff code | 8542.3190 |
| Made in | Switzerland |

† These specifications are for the current hardware revision. Specifications for earlier revisions may differ.

‡ The extended temperature range is defined based on components specifications and has been tested during a limited duration (1h). When using the device in harsh environments for a long period of time, we strongly advise to run extensive tests before going to production.




All dimensions are in mm
Toutes les dimensions sont en mm

# Yocto-GPS

A4

Scale
2:1
Echelle