

Yocto-3D, User's guide

Table of contents

1. Introduction	1
1.1. Prerequisites	2
1.2. Optional accessories	3
2. Presentation	5
2.1. Common elements	5
2.2. Specific elements	6
3. The Yocto-3D functions	7
3.1. The 3D accelerometer	7
3.2. The 3D magnetometer	7
3.3. The 3D gyroscope	7
3.4. The reference frame	7
3.6. The magnetic compass	9
3.7. The estimated attitude quaternion	9
4. First steps	11
4.1. Localization	11
4.2. Test of the module	11
4.3. Configuration	12
5. Assembly and connections	15
5.1. Fixing	15
5.2. Moving the sensor away	16
5.3. USB power distribution	17
6. Programming, general concepts	19
6.1. Programming paradigm	19
6.2. The Yocto-3D module	20
6.3. Module control interface	23
6.4. Accelerometer function interface	24
6.5. Magnetometer function interface	25
6.6. Gyroscope function interface	26
6.7. Reference frame configuration	27
6.8. Tilt function interface	28

6.9. Compass function interface	29
6.10. DataLogger function interface	30
6.11. What interface: Native, DLL or Service ?	30
6.12. Programming, where to start?	33
7. Using the Yocto-3D in command line	35
7.1. Installing	35
7.2. Use: general description	35
7.3. Control of the Tilt function	36
7.4. Control of the module part	36
7.5. Limitations	37
8. Using Yocto-3D with Javascript	39
8.1. Getting ready	39
8.2. Control of the Tilt function	39
8.3. Control of the module part	41
8.4. Error handling	44
9. Using Yocto-3D with PHP	47
9.1. Getting ready	47
9.2. Control of the Tilt function	47
9.3. Control of the module part	50
9.4. HTTP callback API and NAT filters	52
9.5. Error handling	55
10. Using Yocto-3D with C++	57
10.1. Control of the Tilt function	57
10.2. Control of the module part	60
10.3. Error handling	62
10.4. Integration variants for the C++ Yoctopuce library	63
11. Using Yocto-3D with Objective-C	65
11.1. Control of the Tilt function	65
11.2. Control of the module part	67
11.3. Error handling	69
12. Using Yocto-3D with Visual Basic .NET	71
12.1. Installation	71
12.2. Using the Yoctopuce API in a Visual Basic project	71
12.3. Control of the Tilt function	72
12.4. Control of the module part	74
12.5. Error handling	76
13. Using Yocto-3D with C#	79
13.1. Installation	79
13.2. Using the Yoctopuce API in a Visual C# project	79
13.3. Control of the Tilt function	80
13.4. Control of the module part	82
13.5. Error handling	84
14. Using Yocto-3D with Delphi	87
14.1. Preparation	87
14.2. Control of the Tilt function	87
14.3. Control of the module part	89

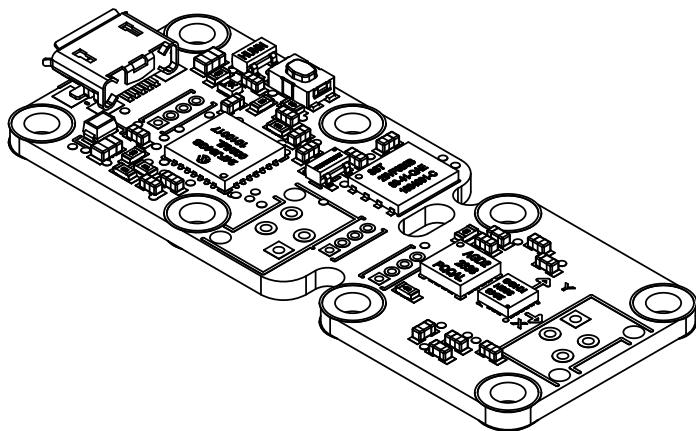
14.4. Error handling	92
15. Using the Yocto-3D with Python	93
15.1. Source files	93
15.2. Dynamic library	93
15.3. Control of the Tilt function	93
15.4. Control of the module part	95
15.5. Error handling	97
16. Using the Yocto-3D with Java	99
16.1. Getting ready	99
16.2. Control of the Tilt function	99
16.3. Control of the module part	101
16.4. Error handling	104
17. Using the Yocto-3D with Android	105
17.1. Native access and VirtualHub	105
17.2. Getting ready	105
17.3. Compatibility	105
17.4. Activating the USB port under Android	106
17.5. Control of the Tilt function	108
17.6. Control of the module part	111
17.7. Error handling	115
18. Using with unsupported languages	117
18.1. Command line	117
18.2. VirtualHub and HTTP GET	117
18.3. Using dynamic libraries	119
18.4. Porting the high level library	122
19. Advanced programming	123
19.1. Event programming	123
19.2. The data logger	126
19.3. Sensor calibration	128
20. High-level API Reference	133
20.1. General functions	134
20.2. Module control interface	159
20.3. Accelerometer function interface	209
20.4. Magnetometer function interface	255
20.5. Reference frame configuration	301
20.6. Tilt function interface	341
20.7. Compass function interface	384
20.8. Gyroscope function interface	428
20.9. DataLogger function interface	483
20.10. Recorded data sequence	521
20.11. Measured value	534
21. Troubleshooting	541
21.1. Linux and USB	541
21.2. ARM Platforms: HF and EL	542
22. Characteristics	543
Blueprint	545

1. Introduction

The Yocto-3D is a 51x20mm module which contains an accelerometer, a gyroscope, and a magnetometer. The three sensors are three-dimensional. The Yocto-3D uses these three sensors to measure the inclination with regards to the horizontal plane (tilt sensor), the magnetic heading (tilt-compensated compass), the acceleration, and the magnetic field to which the module is subjected as well as the angular velocity on each axis.

Moreover, the module is able to instantly estimate its orientation by combining the integration of gyroscopic measures with the static orientation measurements. The device attitude is provided directly as a quaternion, with an optional conversion into aeronautical angles in the API (using Tait-Bryan's angles: roll, pitch, and heading). No extra math is required in the application to get this information.

A flash memory is provided in the module, enabling it to autonomously store the measures for later retrieval.



The Yocto-3D module

Yoctopuce thanks you for buying this Yocto-3D 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-3D is easy to install anywhere and easy to drive from a maximum of programming languages. If you are nevertheless disappointed with this module, do not hesitate to contact Yoctopuce support¹.

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.

¹ support@yoctopuce.com

1.1. Prerequisites

In order to use your Yocto-3D 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, Mac OS X, Linux, and Android. Yoctopuce modules do not require installing any specific system driver, as they leverage the standard HID driver² provided with every operating system.

Windows versions currently supported are: Windows XP, Windows 2003, Windows Vista, Windows 7 and Windows 8.1. Both 32 bit and 64 bit versions are supported. Yoctopuce is frequently testing its modules on Windows XP and Windows 7.

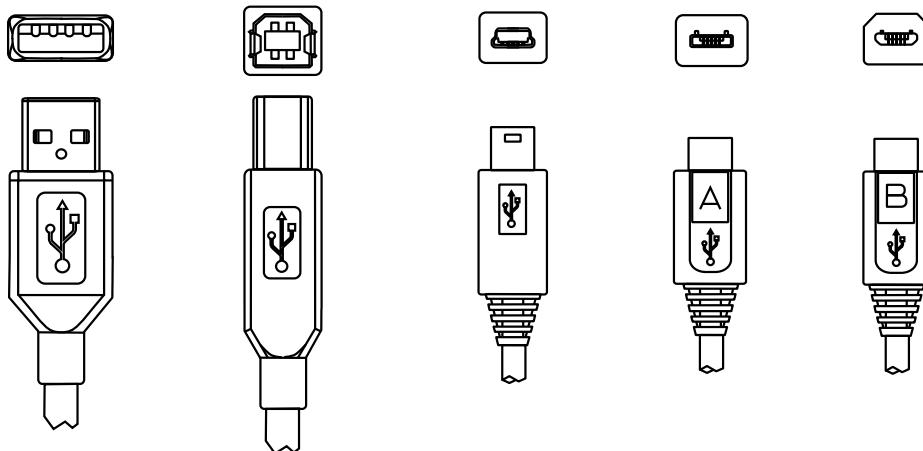
Mac OS X versions currently supported are: 10.6 (Snow Leopard), Mac OS X 10.7 (Lion), 10.8 (Mountain Lion), and 10.9 (Maverick). Yoctopuce is frequently testing its modules on Mac OS X 10.9 and 10.7.

Linux kernels currently supported are the 2.6 branch and the 3.0 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 2.6.

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 4.x on a Nexus 7 and a Samsung Galaxy S3 with the Java for Android library.

A USB cable, type A-micro B

USB 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.



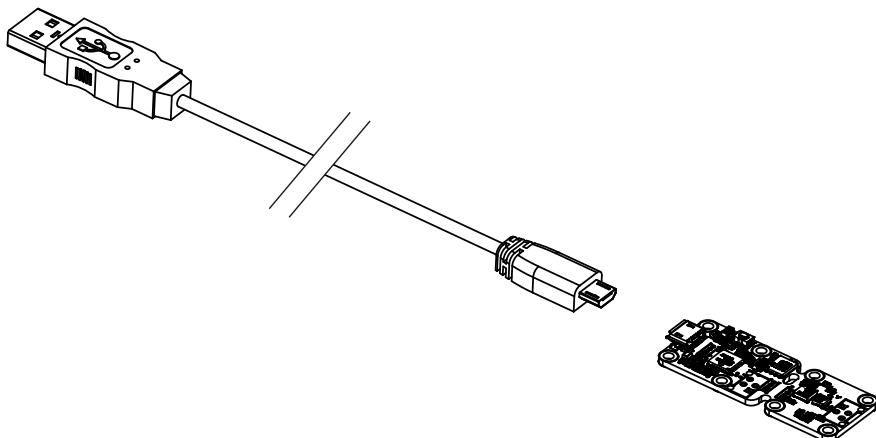
The most common USB 2 connectors: A, B, Mini B, Micro A, Micro B.³

To connect your Yocto-3D module to a computer, you need a USB 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 A to micro B*

² The HID driver is the one that takes care of the mouse, the keyboard, etc.

³ Although they existed for some time, Mini A connectors are not available anymore http://www.usb.org/developers/Deprecation_Announcement_052507.pdf

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-3D module with a USB cable of type A - micro B.

If you insert a USB hub between the computer and the Yocto-3D 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.

1.2. Optional accessories

The accessories below are not necessary to use the Yocto-3D 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-3D 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 and YoctoHub-Wireless

You can add network connectivity to your Yocto-3D, thanks to the YoctoHub-Ethernet and the YoctoHub-Wireless. The YoctoHub-Ethernet provides Ethernet connectivity and the YoctoHub-Wireless provides WiFi connectivity. Both can drive up to three devices and behave exactly like a regular computer running a *VirtualHub*.

Solid copper ribbon cable

If you intend to move the Yocto-3D module sensor away using a ribbon cable directly soldered to the printed circuit board, consider using solid copper ribbon cable: it is much easier to solder. In any case, you will need cable with 1.27mm pitch.

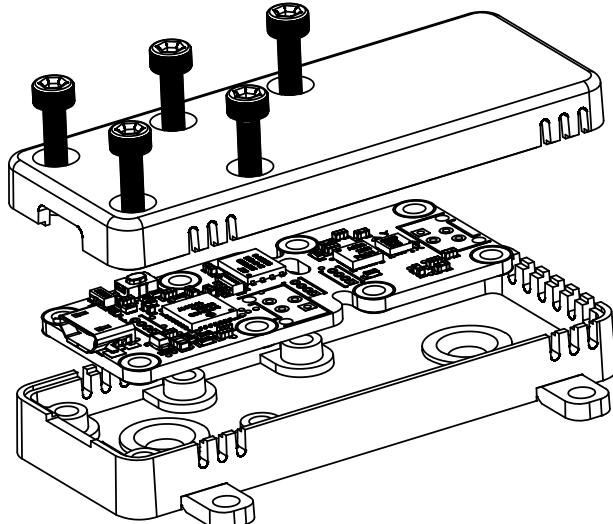
The same cable can be used to solder a cable directly between the Yocto-3D module and a micro-USB hub to save on the space used by USB cables.

Picoflex connectors and flexible ribbon cable

If you intend to move the sensor component away from the rest of the Yocto-3D module using a pluggable cable, you will need 4-wire ribbon cable of 1.27mm pitch, and Picoflex connectors.⁴ You can find more details on this topic in the chapter about assembly and connections.

Enclosures

Your Yocto-3D 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. More details are available on the Yoctopuce web site⁵. The suggested enclosure model for your Yocto-3D is the YoctoBox-3D-Transp. This enclosure does not contain magnets and uses non-ferrous metal screws and bolts, in order to avoid perturbing the Yocto-3D magnetometer.

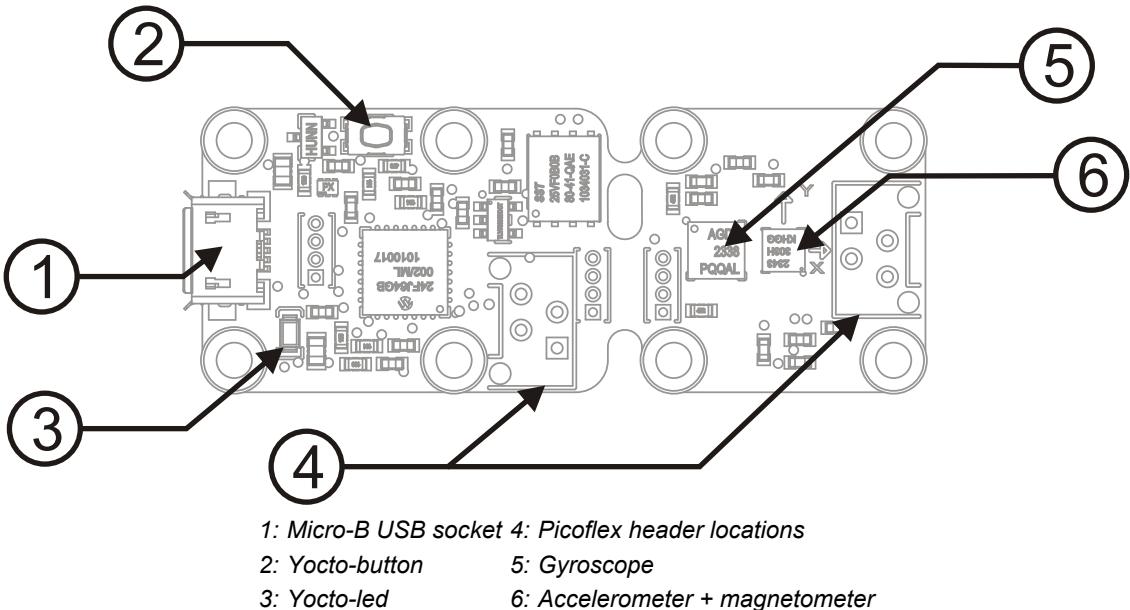


You can install your Yocto-3D in an optional enclosure

⁴ Header Molex ref 90325-3004 or 90325-0004, available from most electronic components suppliers (www.mouser.com, www.digikey.com, www.farnell.com, www.distrelec.ch...). To be used with connectors ref 90327-3304 or 90327-0304.

⁵ <http://www.yoctopuce.com/EN/products/category/enclosures>

2. Presentation



2.1. Common elements

All Yocto-modules share a number of common functionalities.

USB connector

Yoctopuce modules all come with a micro-B USB socket. The corresponding cables are not the most common, but the sockets are the smallest available.

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.

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 morse¹. 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 support².

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-3D modules, this number starts with Y3DMK001. 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 sensors

The Yocto-3D uses a STMicroElectronics LSM303D sensor. It is a 3D accelerometer coupled with a 3D magnetometer. An acceleration above 3000g for more than 0.5 [s], an acceleration of 10000g for more than 0.1 [s], or a magnetic field of more than 1000 gauss would damage this sensor.

The Yocto-3D also uses an STMicroElectronics L3GD20 3D gyroscope. A 10000g acceleration for more than 0.1 [s] would damage this sensor.

¹ short-short-short long-long-long short-short-short

² support@yoctopuce.com

3. The Yocto-3D functions

The Yocto-3D has many distinct functions, providing various interfaces to its inertial sensors as may be required by the target application.

3.1. The 3D accelerometer

This function shows the acceleration to which the module is submitted, in multiples of the gravitational acceleration (g). The main value is the norm of the acceleration vector, but you can obtain X, Y, and Z components individually. The maximal measured value is of about 16g.

To obtain a very precise value taking into account the gravitational acceleration at the location where the module is used, you can perform a semi-automatic calibration of the accelerometer.

3.2. The 3D magnetometer

This function shows the magnetic field intensity to which the module is subjected, in Gauss. The main value is the field vector, but you can obtain the X, Y, and Z values individually. To be as precise as possible to detect the earth magnetic field, the magnetometer is limited to +/- 2 Gauss.

Beware, this function is easily perturbed in the proximity of magnets, electro-magnetic fields, or even simple ferromagnetic objects (such as steel screws) which modify the field lines.

3.3. The 3D gyroscope

This function shows the angular velocity of the device, in degrees per second. The main value is the norm of the angular velocity vector, but you can also obtain the angular velocity around individual axis X, Y, and Z. The maximal measured value is about 2000 degrees per second.

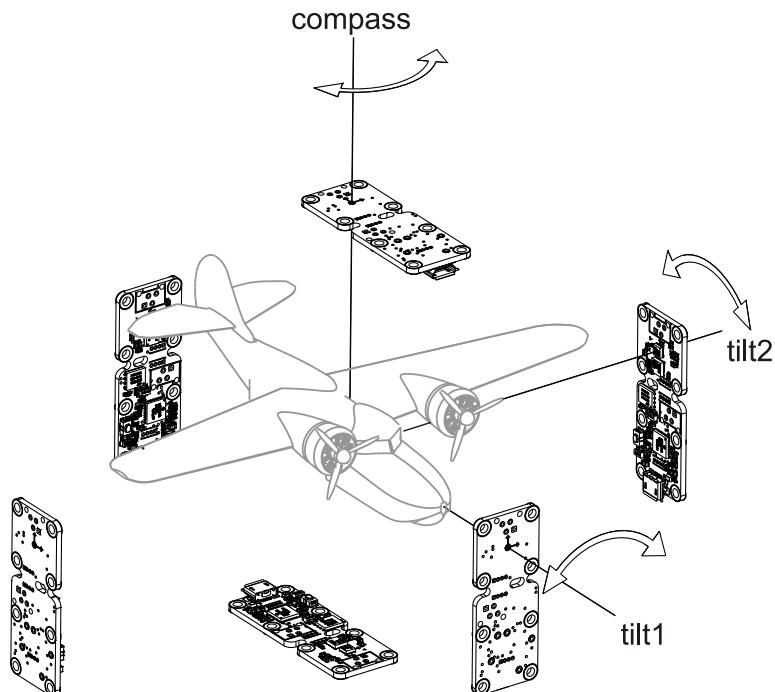
In order to counter the natural drift of the gyroscope, the Yocto-3D automatically zeroes the measured value when the device detects that it is not moving (acceleration vector staying still and around 1g for several tenths of seconds at least).

3.4. The reference frame

You can freely choose the reference frame in which the Yocto-3D is going to work, that is the orientation in which it is assembled with regards to the natural movement direction. Indeed, the module orientation intervenes directly in the tilt measure computations (which are measured with regards to the horizontal plane), as well as for angle measures. In all, there are 24 orthogonal

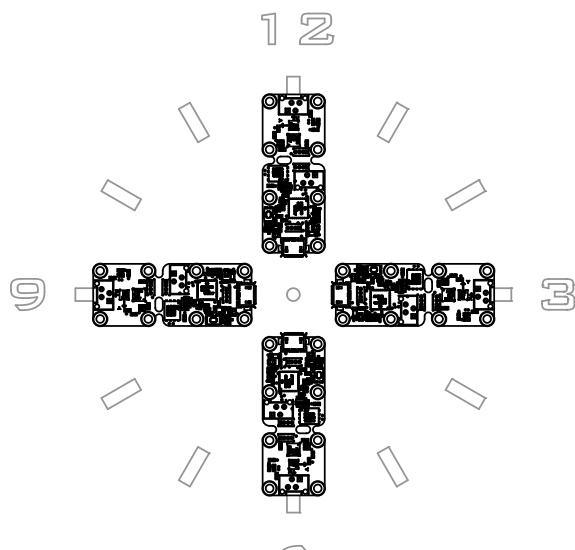
references you can select from. They can be simplified into 6 groups of positions (corresponding to the 6 sides of a cube), each including 4 distinct references for the 4 possible 90° rotations around the module Z axis.

Let us imagine that the Yocto-3D is assembled into a plane. It can be fixed on the left wall (LEFT), on the right wall (RIGHT), on the floor (BOTTOM), on the ceiling (TOP), at the front (FRONT), or at the rear (REAR).



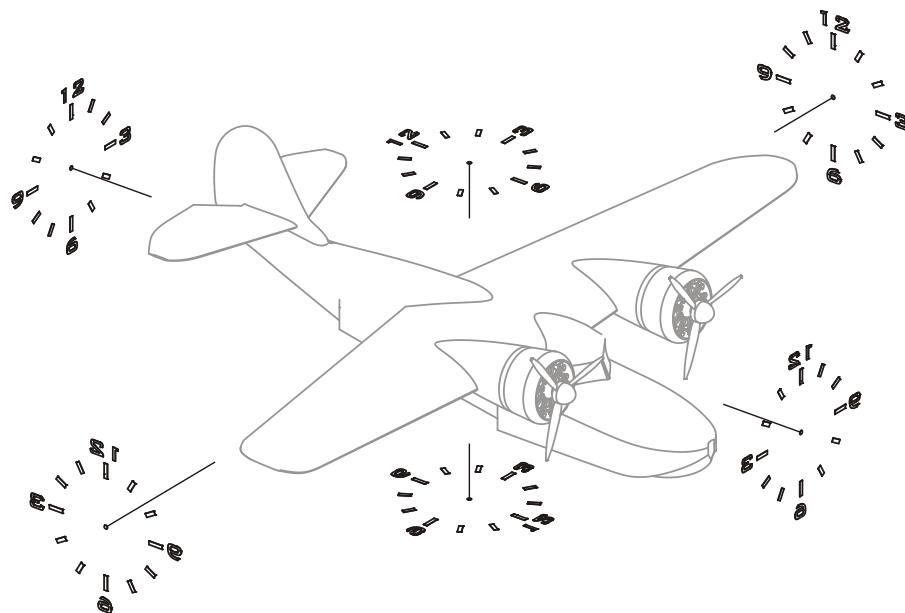
The module 6 assembly positions

For each assembly position, there are 4 orthogonal orientations, which are defined by analogy with a clock: 12:00, 3:00, 6:00, and 9:00.



The 4 assembly orientations

The orientation of each of the 6 clocks is defined in a natural way with regards to what the pilot of the plane would see.



Orientation of the six clocks

Make sure to configure the Yocto-3D orientation after it is assembled, as all functions listed below in this sections depend on this settings. Do not forget to save this configuration in the module flash memory. Otherwise, the Yocto-3D will take the default configuration (BOTTOM, TWELVE) the next time it starts.

3.5. The 2D inclinometer (tilt sensor)

The Yocto-3D can measure the tilt with regards to the horizontal plane, with the help of a gravity measure. You can configure the choice of the reference horizontal plane in the `refFrame` function described above. By default, it is the module X/Y plane.

The `tilt1` function corresponds to roll, while the `tilt2` function corresponds to pitch. Roll is a positive angle when the left side of the module rises. And pitch is a positive angle when the front of the module rises.

3.6. The magnetic compass

This function provides the module bearing, such as determined thanks to the earth magnetic field (as with a compass). The bearing is a angle in degrees, from 0° to 360°, clockwise. The magnetic field measure is compensated to take into account the module tilt. The module does not need, therefore, to be perfectly horizontal for the value to be correct.

By default, bearing is computed on the module X/Y plane, but this plan can be changed by modifying the orientation defined with the `refFrame` function. Likewise, by default, 0° corresponds to the magnetic North, but it is possible to configure a different reference angle, for instance to compensate for the local magnetic declination and to point to the geographic North.

Beware, this function is easily perturbed in the proximity of magnets, electro-magnetic fields, or even simple ferromagnetic objects (such as steel screws) which modify the field lines. To obtain a value as precise as possible, you can perform a semi-automatic calibration of the magnetometer.

3.7. The estimated attitude quaternion

The four functions `qt1` to `qt4` correspond to the four components of the hypercomplex quaternion (also known as w, x, y and z) describing the estimated attitude of the device, based on a 95Hz gyroscopic integration. To counter the natural drift of the integration process, the Yocto-3D combines it with tilt and compass measurements using a complimentary filter.

The `YGyro` class of the programming library provides a simple API to access the quaternion, including its conversion into Tait-Bryan's roll, pitch, and heading angles).

4. First steps

When reading this chapter, your Yocto-3D 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* software¹. 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 browser². 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.

Serial	Logical Name	Description	Action
VIRTHUB0-1521ca755		VirtualHub	configure view log file
Y3DMK001-1D21A		Yocto-3D	configure view log file beacon

Module list as displayed in your web browser.

4.1. Localization

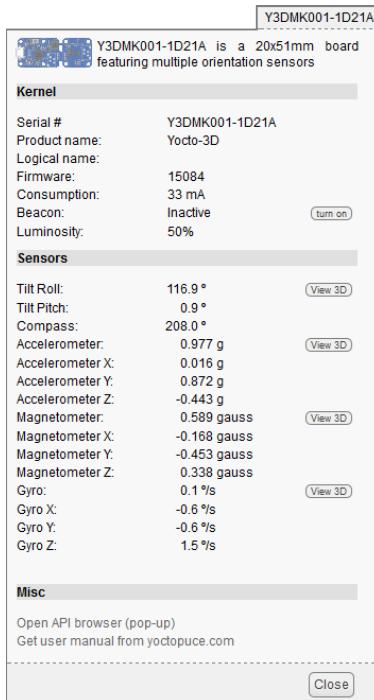
You can then physically localize each of the displayed modules by clicking on the **beacon** button. This puts the Yocto-led of the corresponding module in Yocto-beacon mode. It starts flashing, which allows you to easily localize it. The second effect is to display a little blue circle on the screen.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

4.2. 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-3D.

¹ www.yoctopuce.com/EN/virtualhub.php

² The interface was tested on FireFox 3+, IE 6+, Safari, and Chrome.

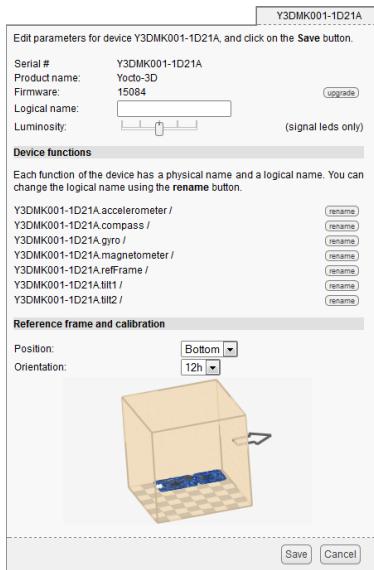


Properties of the Yocto-3D module.

This window allows you to play with your module to check how it is working. Values measured by the Yocto-3D are indeed displayed in real time, both as numerical values and as a graph.

4.3. Configuration

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



Yocto-3D module configuration.

Firmware

The module firmware can easily be updated with the help of the interface. To do so, you must beforehand have the adequate firmware on your local disk. 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* software³ in command line⁴.

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.

The Yocto-3D module has 9 functions. Simply click on the corresponding "rename" buttons to assign them new logical names.

You can also configure the reference frame used by the module for the tilt sensors, the compass, and the gyroscopic estimate of the module orientation.

³ www.yoctopuce.com/EN/virtualhub.php

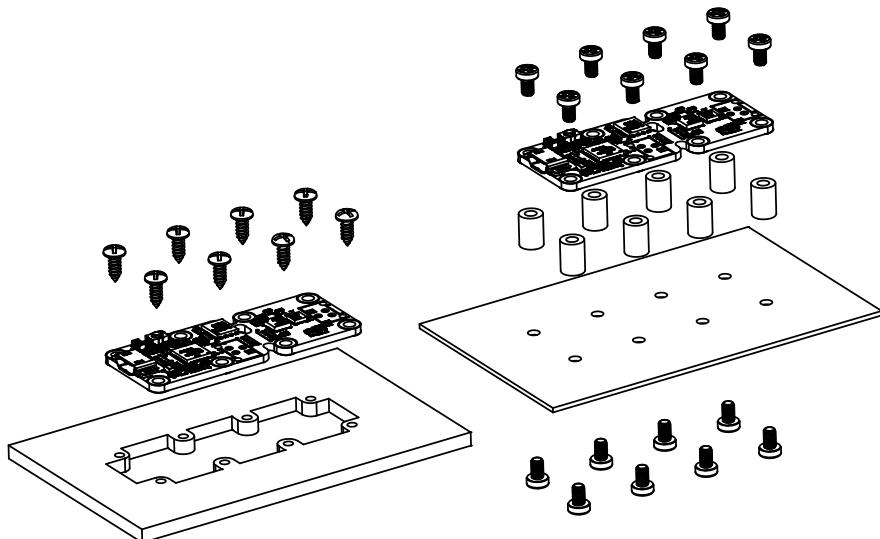
⁴ More information available in the virtual hub documentation

5. Assembly and connections

This chapter provides important information regarding the use of the Yocto-3D module in real-world situations. Make sure to read it carefully before going too far into your project if you want to avoid pitfalls.

5.1. 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

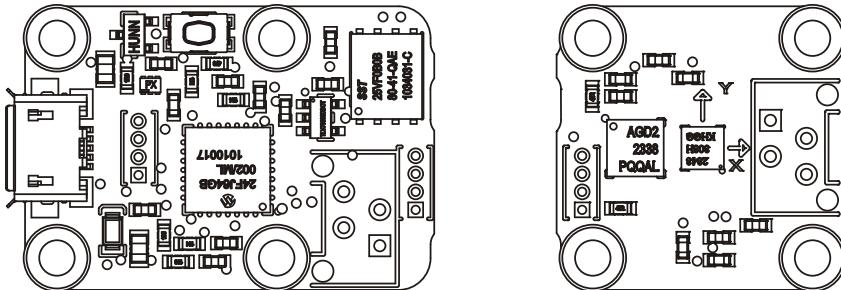
The Yocto-3D module contains 2.5mm assembly holes. You can use these holes for screws. The screw head diameter must not be larger than 4.5mm or they will damage the module circuits. Make sure that the lower surface of the module is not in contact with the support. We recommend using spacers, but other 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 you 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 packaging tape should be enough for electric insulation.

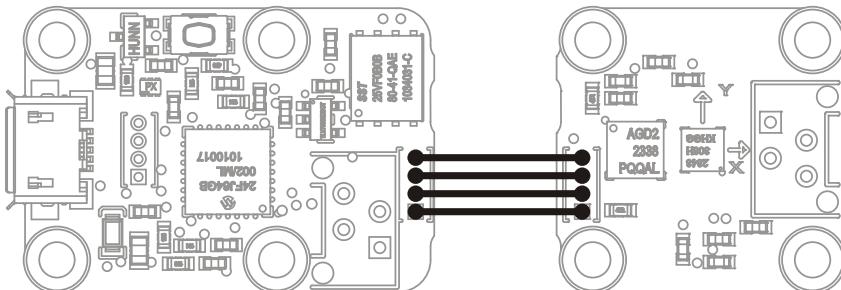
The use of mounting elements made of ferrous material, such as steel screws, is likely to prevent the magnetometer and the compass from working correctly.

5.2. Moving the sensor away

The Yocto-3D module is designed so that you can split it into two parts, allowing you to move away the sensor from the command sub-module. You can split the module by simply breaking the circuit. However, you can obtain better results if you use a good pincer, or cutting pliers. When you have split the sub-modules, you can sandpaper the protruding parts without risk.

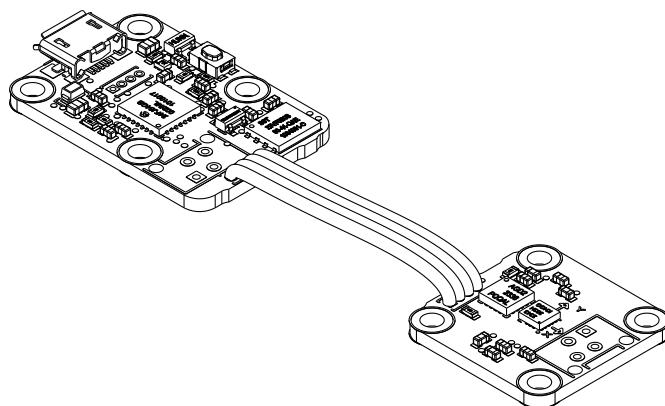


The Yocto-3D module is designed so that you can split it into two parts.



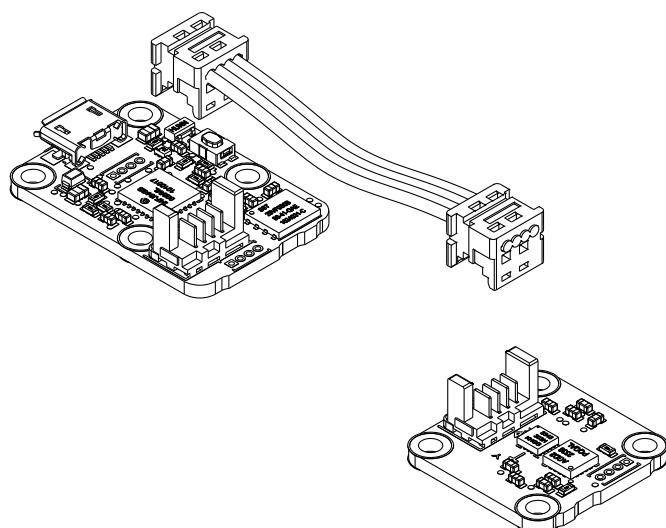
Wiring under the sub-modules once separated.

Once the module is split into two, you must rewire the sub-modules. You can connect the sub-modules by soldering simple electric wires, but you can obtain a better result with 1.27 pitch ribbon cable. Consider using solid copper cables, rather than threaded ones: solid copper cables are somewhat less flexible, but much easier to solder.



Moving the sensor away with a ribbon cable.

You can also use ribbon cable with Picoflex connectors. You will obtain a slightly larger system, but the Picoflex headers are much easier to solder than ribbon cable. More over, the result can be disassembled.



Moving the sensor away with PicoFlex connectors.

Make sure to keep the sensor part away from ferromagnetic elements: a simple steel screw can considerably alter the magnetometer values, and therefore the compass.

Warning: divisible Yoctopuce modules very often have very similar connection systems. Nevertheless, sub-modules from different models are not all compatible. If you connect your Yocto-3D sub-module to another type of module, such as a Yocto-Temperature for instance, it will not work, and you run a high risk of damaging your equipment.

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

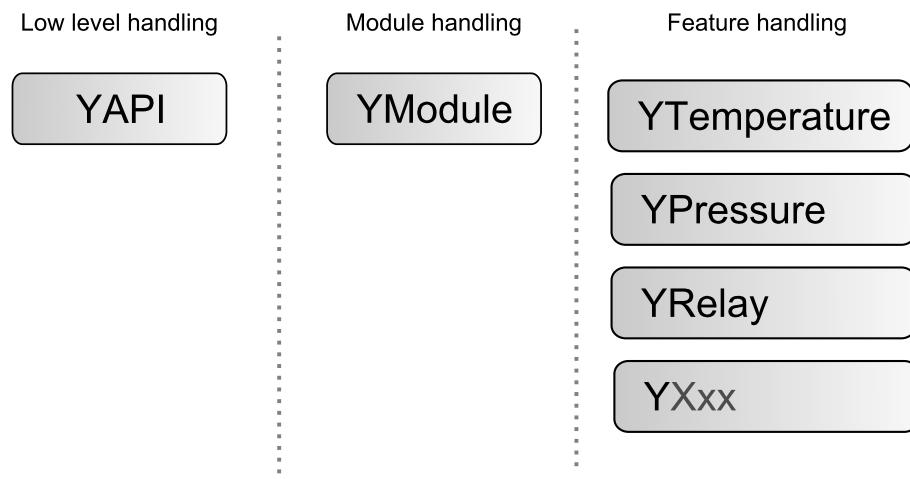
6. Programming, general concepts

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-3D 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.

6.1. 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.

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-3D 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.

6.2. The Yocto-3D module

The Yocto-3D device provides multiple measures. Raw sensor values can be obtained through the accelerometer, magnetometer and gyro interfaces. Computed inclination relative to the horizontal plane can be obtained through the tilt1 and tilt2 interfaces, as well as heading relative to the horizontal magnetic field. The device also provides an estimate of its 3D orientation based on the mathematical fusion of all sensors.

module : Module

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

accelerometer : Accelerometer

attribute	type	modifiable ?
logicalName	String	modifiable
advertisedValue	String	read-only
unit	String	read-only
currentValue	Fixed-point number	read-only
lowestValue	Fixed-point number	modifiable
highestValue	Fixed-point number	modifiable
currentRawValue	Fixed-point number	read-only
logFrequency	Frequency	modifiable
reportFrequency	Frequency	modifiable
calibrationParam	Calibration parameters	modifiable
resolution	Fixed-point number	modifiable
xValue	Fixed-point number	read-only
yValue	Fixed-point number	read-only
zValue	Fixed-point number	read-only
gravityCancellation	On/Off	modifiable

magnetometer : Magnetometer

attribute	type	modifiable ?
logicalName	String	modifiable
advertisedValue	String	read-only
unit	String	read-only
currentValue	Fixed-point number	read-only
lowestValue	Fixed-point number	modifiable
highestValue	Fixed-point number	modifiable
currentRawValue	Fixed-point number	read-only
logFrequency	Frequency	modifiable
reportFrequency	Frequency	modifiable
calibrationParam	Calibration parameters	modifiable
resolution	Fixed-point number	modifiable
xValue	Fixed-point number	read-only
yValue	Fixed-point number	read-only
zValue	Fixed-point number	read-only

gyro : Gyro

attribute	type	modifiable ?
logicalName	String	modifiable
advertisedValue	String	read-only
unit	String	read-only
currentValue	Fixed-point number	read-only
lowestValue	Fixed-point number	modifiable

highestValue	Fixed-point number	modifiable
currentRawValue	Fixed-point number	read-only
logFrequency	Frequency	modifiable
reportFrequency	Frequency	modifiable
calibrationParam	Calibration parameters	modifiable
resolution	Fixed-point number	modifiable
xValue	Fixed-point number	read-only
yValue	Fixed-point number	read-only
zValue	Fixed-point number	read-only

refFrame : RefFrame

attribute	type	modifiable ?
logicalName	String	modifiable
advertisedValue	String	read-only
mountPos	Integer	modifiable
bearing	Fixed-point number	modifiable
calibrationParam	Calibration parameters	modifiable

tilt1 : Tilt**tilt2 : Tilt**

attribute	type	modifiable ?
logicalName	String	modifiable
advertisedValue	String	read-only
unit	String	read-only
currentValue	Fixed-point number	read-only
lowestValue	Fixed-point number	modifiable
highestValue	Fixed-point number	modifiable
currentRawValue	Fixed-point number	read-only
logFrequency	Frequency	modifiable
reportFrequency	Frequency	modifiable
calibrationParam	Calibration parameters	modifiable
resolution	Fixed-point number	modifiable
axis	Rotation axis	read-only

compass : Compass

attribute	type	modifiable ?
logicalName	String	modifiable
advertisedValue	String	read-only
unit	String	read-only
currentValue	Fixed-point number	read-only
lowestValue	Fixed-point number	modifiable
highestValue	Fixed-point number	modifiable
currentRawValue	Fixed-point number	read-only
logFrequency	Frequency	modifiable
reportFrequency	Frequency	modifiable
calibrationParam	Calibration parameters	modifiable
resolution	Fixed-point number	modifiable
axis	Rotation axis	read-only
magneticHeading	Fixed-point number	read-only

qt1 : Qt**qt2 : Qt****qt3 : Qt****qt4 : Qt**

attribute	type	modifiable ?
logicalName	String	modifiable
advertisedValue	String	read-only
unit	String	read-only

currentValue	Fixed-point number	read-only
lowestValue	Fixed-point number	modifiable
highestValue	Fixed-point number	modifiable
currentRawValue	Fixed-point number	read-only
logFrequency	Frequency	modifiable
reportFrequency	Frequency	modifiable
calibrationParam	Calibration parameters	modifiable
resolution	Fixed-point number	modifiable

dataLogger : DataLogger

attribute	type	modifiable ?
logicalName	String	modifiable
advertisedValue	String	read-only
currentRunIndex	Integer	read-only
timeUTC	UTC time	modifiable
recording	On/Off	modifiable
autoStart	On/Off	modifiable
beaconDriven	On/Off	modifiable
clearHistory	Boolean	modifiable

6.3. Module control interface

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

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-3D module, this serial number always starts with Y3DMK001. 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 a 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 59 at the factory.

productRelease

Release number of the module hardware, preprogrammed at the factory.

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.

6.4. Accelerometer function interface

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

logicalName

Character string containing the logical name of the accelerometer, initially empty. This attribute can be modified at will by the user. Once initialized to a non-empty value, it can be used to access the accelerometer directly. If two accelerometers 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 accelerometer, that is automatically advertised up to the parent hub. For an accelerometer, the advertised value is the current value of the acceleration.

unit

Short character string representing the measuring unit for the acceleration.

currentValue

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

lowestValue

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

highestValue

Maximal value of the acceleration, in g, 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.

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).

xValue

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

yValue

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

zValue

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

6.5. Magnetometer function interface

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

logicalName

Character string containing the logical name of the magnetometer, 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 magnetometer directly. If two magnetometers 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 magnetometer, that is automatically advertised up to the parent hub. For a magnetometer, the advertised value is the current value of the magnetic field.

unit

Short character string representing the measuring unit for the magnetic field.

currentValue

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

lowestValue

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

highestValue

Maximal value of the magnetic field, in mT, 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.

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).

xValue

X component of the magnetic field, as a floating point number.

yValue

Y component of the magnetic field, as a floating point number.

zValue

Z component of the magnetic field, as a floating point number.

6.6. Gyroscope function interface

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

logicalName

Character string containing the logical name of the gyroscope, initially empty. This attribute can be modified at will by the user. Once initialized to a non-empty value, it can be used to access the gyroscope directly. If two gyroscopes 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 gyroscope, that is automatically advertised up to the parent hub. For a gyroscope, the advertised value is the current value of the angular velocity.

unit

Short character string representing the measuring unit for the angular velocity.

currentValue

Current value of the angular velocity, in degrees per second, as a floating point number.

lowestValue

Minimal value of the angular velocity, in degrees per second, as a floating point number.

highestValue

Maximal value of the angular velocity, in degrees per second, 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.

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).

xValue

Angular velocity around the X axis of the device, as a floating point number.

yValue

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

zValue

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

6.7. Reference frame configuration

This class is used to setup the base orientation of the Yocto-3D, so that the orientation functions, relative to the earth surface plane, use the proper reference frame. The class also implements a tridimensional sensor calibration process, which can compensate for local variations of standard gravity and improve the precision of the tilt sensors.

logicalName

Character string containing the logical name of the reference frame, 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 reference frame directly. If two reference frames 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 reference frame, that is automatically advertised up to the parent hub. For a reference frame, the advertised value is its the ordered list of axis device corresponding to roll, pitch and heading.

mountPos

Installation position of the device, defining the reference frame for the compass and the pitch/roll tilt sensors.

bearing

Reference bearing used by the compass.

calibrationParam

Low-level calibration parameters for the accelerometer and magnetometer used by the compass and pitch/roll tilt sensors.

6.8. Tilt function interface

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

logicalName

Character string containing the logical name of the tilt sensor, initially empty. This attribute can be modified at will by the user. Once initialized to a non-empty value, it can be used to access the tilt sensor directly. If two tilt 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 tilt sensor, that is automatically advertised up to the parent hub. For a tilt sensor, the advertised value is the current value of the inclination.

unit

Short character string representing the measuring unit for the inclination.

currentValue

Current value of the inclination, in degrees, as a floating point number.

lowestValue

Minimal value of the inclination, in degrees, as a floating point number.

highestValue

Maximal value of the inclination, in degrees, 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.

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).

axis

Function working axis for the tilt sensor.

6.9. Compass function interface

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

logicalName

Character string containing the logical name of the compass, 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 compass directly. If two compasses 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 compass, that is automatically advertised up to the parent hub. For a compass, the advertised value is the current value of the relative bearing.

unit

Short character string representing the measuring unit for the relative bearing.

currentValue

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

lowestValue

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

highestValue

Maximal value of the relative bearing, in degrees, 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.

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).

axis

Function working axis for the compass.

magneticHeading

Magnetic heading (regardless of the configured bearing)

6.10. DataLogger function interface

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

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 a 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.

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.

beaconDriven

Synchronize the state 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 state of the localization beacon is updated. Note: when this attribute is set the localization beacon pulses slower than usual.

clearHistory

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

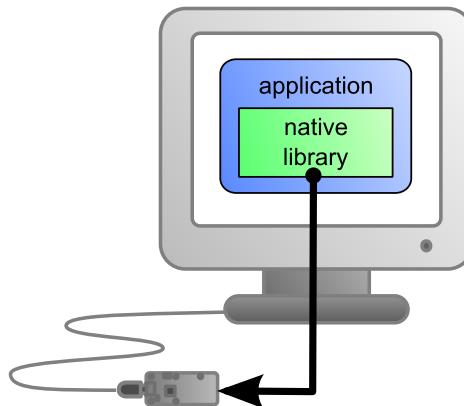
6.11. What interface: Native, DLL or Service ?

There are several methods to control your 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

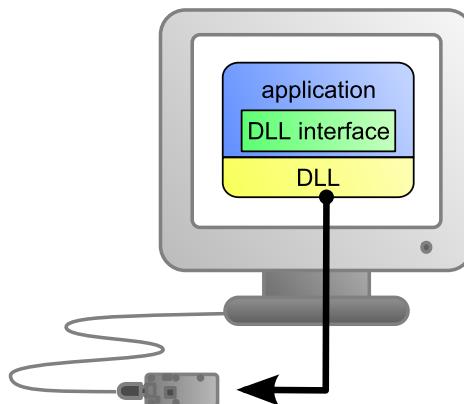
user then only needs to plug the USB cable and run your software for everything to work. Unfortunately, this method is not always available or even possible.



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.

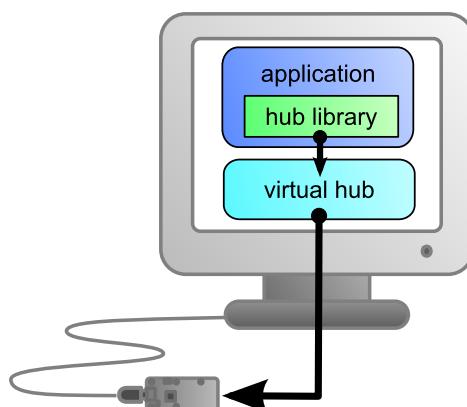


The application uses the DLL to natively control the locally connected module

Control by service

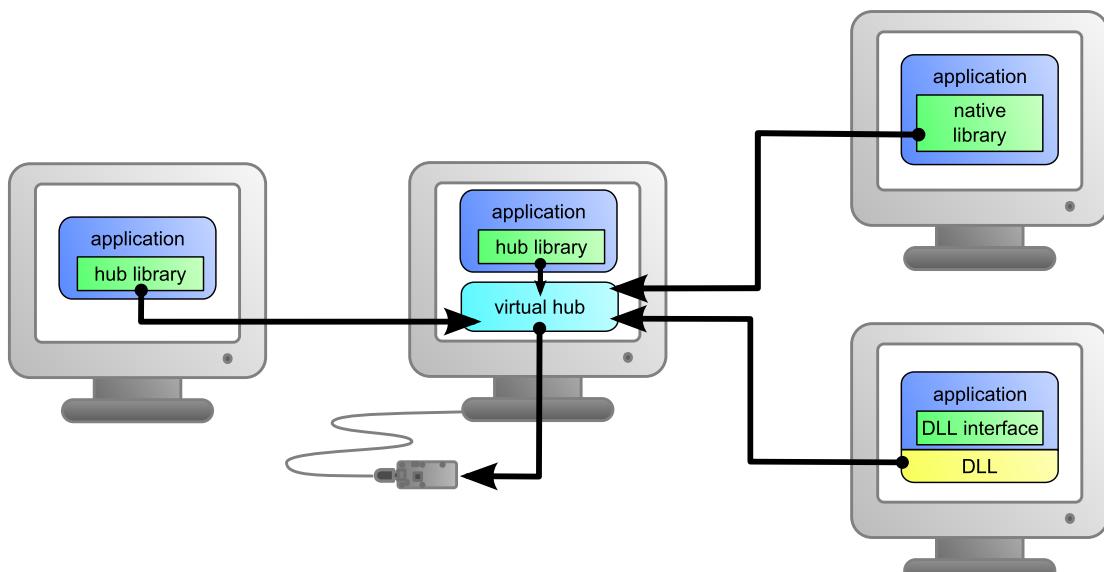
Some languages do simply not allow you to easily gain access to the hardware layers of the machine. It is the case for Javascript, for instance. To deal with this case, Yoctopuce provides a solution in the form of a small piece of software called *Virtual Hub*¹. It can access the modules, and your application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

¹ www.yoctopuce.com/EN/virtualhub.php



The application connects itself to the virtual hub 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 virtual hubs.



When a virtual hub 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	Virtual hub
C++	.	.	.
Objective-C	.	-	.
Delphi	-	.	.
Python	-	.	.
VisualBasic .Net	-	.	.
C# .Net	-	.	.
Javascript	-	-	.
Node.js	-	-	.
PHP	-	-	.
Java	-	-	.
Java for Android	.	-	.
Command line	.	-	.

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.

6.12. Programming, where to start?

At this point of the user's guide, you should know the main theoretical points of your Yocto-3D. It is now time to practice. You must download the Yoctopuce library for your favorite programming language from the Yoctopuce web site². 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.

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-3D.

² <http://www.yoctopuce.com/EN/libraries.php>

7. Using the Yocto-3D in command line

When you want to perform a punctual operation on your Yocto-3D, 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 provided¹.

7.1. Installing

Download the command line API². 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-3D, open a shell, and start working by typing for example:

```
C:\>YTilt any get_currentValue
```

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.

7.2. 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 commas 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.

7.3. Control of the Tilt function

To control the Tilt function of your Yocto-3D, you need the YTilt executable file.

For instance, you can launch:

```
C:\>YTilt any get_currentValue
```

This example uses the "any" target to indicate that we want to work on the first Tilt 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-3D module with the Y3DMK001-123456 serial number which you have called "MyModule", and its tilt1 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:\>YTilt Y3DMK001-123456.tilt1 describe
C:\>YTilt Y3DMK001-123456.MyFunction describe
C:\>YTilt MyModule.tilt1 describe
C:\>YTilt MyModule.MyFunction describe
C:\>YTilt MyFunction describe
```

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

```
C:\>YTilt all describe
```

For more details on the possibilities of the YTilt executable, use:

```
C:\>YTilt /help
```

7.4. 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 Y3DMK001-12346 set_logicalName MonPremierModule
```

```
C:\>YModule Y3DMK001-12346 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 Y3DMK001-12346 set_logicalName MonPremierModule
C:\>YModule Y3DMK001-12346 saveToFlash
```

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

```
C:\>YModule -s Y3DMK001-12346 set_logicalName MonPremierModule
```

Warning: the number of write cycles of the nonvolatile memory of the module is limited. When this limit is reached, nothing guarantees 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.

7.5. 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 `VirtualHub3` 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>

8. Using Yocto-3D with Javascript

Javascript is probably not the first language that comes to mind to control hardware, but its ease of use is a great advantage: with Javascript, you only need a text editor and a web browser to realize your first tests.

Javascript is one of those languages which do not allow you to directly access the hardware layers of your computer. Therefore you need to run the Yoctopuce TCP/IP to USB gateway, named *VirtualHub*, on the machine on which your modules are connected.

8.1. Getting ready

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

- The Javascript programming library¹
- The VirtualHub software² for Windows, Mac OS X or Linux, depending on your OS

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.

8.2. Control of the Tilt function

A few lines of code are enough to use a Yocto-3D. Here is the skeleton of a JavaScript code snippet to use the Tilt function.

```
<SCRIPT type="text/javascript" src="yocto_api.js"></SCRIPT>
<SCRIPT type="text/javascript" src="yocto_tilt.js"></SCRIPT>

// Get access to your device, through the VirtualHub running locally
yRegisterHub('http://127.0.0.1:4444/');
var tilt = yFindTilt("Y3DMK001-123456.tilt1");

// Check that the module is online to handle hot-plug
if(tilt.isOnline())
{
    // Use tilt.get_currentValue(), ...
}
```

Let us look at these lines in more details.

¹ www.yoctopuce.com/EN/libraries.php

² www.yoctopuce.com/EN/virtualhub.php

yocto_api.js and yocto_tilt.js

These two Javascript includes provide access to functions allowing you to manage Yoctopuce modules. `yocto_api.js` must always be included, `yocto_tilt.js` is necessary to manage modules containing a tilt sensor, such as Yocto-3D.

yRegisterHub

The `yRegisterHub` 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.

yFindTilt

The `yFindTilt` function allows you to find a tilt 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-3D module with serial number `Y3DMK001-123456` which you have named "`MyModule`", and for which you have given the `tilt1` function the name "`MyFunction`". The following five calls are strictly equivalent, as long as "`MyFunction`" is defined only once.

```
var tilt = yFindTilt("Y3DMK001-123456.tilt1");
var tilt = yFindTilt("Y3DMK001-123456.MyFunction");
var tilt = yFindTilt("MyModule.tilt1");
var tilt = yFindTilt("MyModule.MyFunction");
var tilt = yFindTilt("MyFunction");
```

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

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `yFindTilt` provides the angle currently measured by the inclinometer. The value returned is a floating number.

yFindCompass and yFindGyro

Functions `yFindCompass`, `yFindGyro`, `yFindMagnetometer` and `yFindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `YFindTilt`.

A real example

Open your preferred text editor³, copy the code sample below, save it in the same directory as the Yoctopuce library files and then use your preferred web browser to access this page. The code is also provided in the directory **Examples/Doc-GettingStarted-Yocto-3D** 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.

The example is coded to be used either from a web server, or directly by opening the file on the local machine. Note that this latest solution does not work with some versions of Internet Explorer, in particular IE 9 on Windows 7, which is not able to open network connections when working on a local file. In order to use Internet Explorer, you should load the example from a web server. No such problem exists with Chrome, Firefox or Safari.

³ If you do not have a text editor, use Notepad rather than Microsoft Word.

If your Yocto-3D 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-3D is connected and where you run the VirtualHub.

```
<HTML>
<HEAD>
<TITLE>Hello World</TITLE>
<SCRIPT type="text/javascript" src="yocto_api.js"></SCRIPT>
<SCRIPT type="text/javascript" src="yocto_tilt.js"></SCRIPT>
<SCRIPT type="text/javascript" src="yocto_compass.js"></SCRIPT>
<SCRIPT type="text/javascript" src="yocto_gyro.js"></SCRIPT>
<SCRIPT type="text/javascript" src="yocto_accelerometer.js"></SCRIPT>

<SCRIPT language='javascript1.5' type='text/JavaScript'>
<!--

// Setup the API to use the VirtualHub on local machine
if(yRegisterHub('http://127.0.0.1:4444/') != YAPI_SUCCESS) {
    alert("Cannot contact VirtualHub on 127.0.0.1");
}

function refresh()
{
    var serial = document.getElementById('serial').value;
    var anytilt,tilt1,tilt2,compass,gyro,accelerometer;

    // try to find a valid serial
    if(serial == '') {
        // or use any connected module suitable for the demo
        anytilt = yFirstTilt();
        if(anytilt) {
            serial = anytilt.module().get_serialNumber();
            document.getElementById('serial').value = serial;
        }
    }

    // retrieve all sensor on the device with that serial
    tilt1      = yFindTilt(serial+".tilt1");
    tilt2      = yFindTilt(serial+".tilt2");
    compass    = yFindCompass(serial+".compass");
    gyro       = yFindGyro(serial+".gyro");
    accelerometer = yFindAccelerometer(serial+".accelerometer");

    if (tilt1.isOnline()) {
        document.getElementById('msg').value = '';
        document.getElementById("tilt1-val").value     = tilt1.get_currentValue();
        document.getElementById("tilt2-val").value     = tilt2.get_currentValue();
        document.getElementById("compass-val").value   = compass.get_currentValue();
        document.getElementById("gyro-val").value      = gyro.get_currentValue();
        document.getElementById("accelerometer-val").value =
accelerometer.get_currentValue();
    } else {
        document.getElementById('msg').value = 'Module not connected';
    }
    setTimeout('refresh()',500);
}
-->
</SCRIPT>
</HEAD>
<BODY onload='refresh();'>
Module to use: <input id='serial'>
<input id='msg' style='color:red;border:none;' readonly><br>
tilt1: <input id='tilt1-val' readonly> &deg;<br>
tilt2: <input id='tilt2-val' readonly></span> &deg;<br>
compass: <input id='compass-val' readonly></span> &deg;<br>
gyro: <input id='gyro-val' readonly></span> &deg;/s<br>
accelerometer: <input id='accelerometer-val' readonly></span> g<br>
</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>
<SCRIPT type="text/javascript" src="yocto_api.js"></SCRIPT>
<SCRIPT language='javascript1.5' type='text/JavaScript'>
<!--
// Use explicit error handling rather than exceptions
yDisableExceptions();

// Setup the API to use the VirtualHub on local machine
if(yRegisterHub('http://127.0.0.1:4444/') != YAPI_SUCCESS) {
    alert("Cannot contact VirtualHub on 127.0.0.1");
}

var module;

function refresh()
{
    var serial = document.getElementById('serial').value;
    if(serial == '') {
        // Detect any connected module suitable for the demo
        module = yFirstModule().nextModule();
        if(module) {
            serial = module.get_serialNumber();
            document.getElementById('serial').value = serial;
        }
    }

    module = yFindModule(serial);
    if(module.isOnline()) {
        document.getElementById('msg').value = '';
        var html = 'serial: '+module.get_serialNumber()+'<br>';
        html += 'logical name: '+module.get_logicalName()+'<br>';
        html += 'luminosity:' +module.get_luminosity()+'%<br>';
        html += 'beacon:';
        if (module.get_beacon() == Y_BEACON_ON)
            html += "ON <a href='javascript:beacon(Y_BEACON_OFF)'>switch off</a><br>";
        else
            html += "OFF <a href='javascript:beacon(Y_BEACON_ON)'>switch on</a><br>";
        html += 'upTime: '+parseInt(module.get_upTime()/1000) +' sec<br>';
        html += 'USB current: '+module.get_usbCurrent() +' mA<br>';
        html += 'logs:<br><pre>' +module.get_lastLogs() + '</pre><br>';
        document.getElementById('data').innerHTML = html;
    } else {
        document.getElementById('msg').value = 'Module not connected';
    }
    setTimeout('refresh()',1000);
}

function beacon(state)
{
    module.set_beacon(state);
}
-->
</SCRIPT>
</HEAD>
<BODY onload='refresh();'>
Module to use: <input id='serial'>
<input id='msg' style='color:red; border:none;' readonly><br>
<span id='data'></span>
</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>Change module settings</TITLE>
<SCRIPT type="text/javascript" src="yocto_api.js"></SCRIPT>
<SCRIPT type='text/JavaScript'>
<!--
// Use explicit error handling rather than exceptions
yDisableExceptions();

// Setup the API to use the VirtualHub on local machine
if(yRegisterHub('http://127.0.0.1:4444') != YAPI_SUCCESS) {
    alert("Cannot contact VirtualHub on 127.0.0.1");
}

var module;

function refresh()
{
    var serial = document.getElementById('serial').value;
    if(serial == '') {
        // Detect any connected module suitable for the demo
        module = yFirstModule().nextModule();
        if(module) {
            serial = module.get_serialNumber();
            document.getElementById('serial').value = serial;
        }
    }

    module = yFindModule(serial);
    if(module.isOnline()) {
        document.getElementById('msg').value = '';
        document.getElementById('curName').value = module.get_logicalName();
    } else {
        document.getElementById('msg').value = 'Module not connected';
    }
    setTimeout('refresh()',1000);
}

function save()
{
    var newname = document.getElementById('newName').value;
    if (!yCheckLogicalName(newname)) {
        alert('invalid logical name');
        return;
    }
    module.set_logicalName(newname);
    module.saveToFlash();
}
-->
</SCRIPT>
</HEAD>
<BODY onload='refresh();'>
Module to use: <input id='serial'>
<input id='msg' style='color:red;border:none;' readonly><br>
Current name: <input id='curName' readonly><br>
New logical name: <input id='newName'>
<a href='javascript:save();'>Save</a>
</BODY>
</HTML>
```

Warning: the number of write cycles of the nonvolatile memory of the module is limited. When this limit is reached, nothing guarantees 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>Modules inventory</TITLE>
<SCRIPT type="text/javascript" src="yocto_api.js"></SCRIPT>
<SCRIPT type='text/JavaScript'>
<!--
// Use explicit error handling rather than exceptions
yDisableExceptions();

// Setup the API to use the VirtualHub on local machine
if(yRegisterHub('http://127.0.0.1:4444/') != YAPI_SUCCESS) {
    alert("Cannot contact VirtualHub on 127.0.0.1");
}

function refresh()
{
    yUpdateDeviceList();

    var htmlcode = '';
    var module = yFirstModule();
    while(module) {
        htmlcode += module.get_serialNumber()
                    +'('+module.get_productName() +")<br>";
        module = module.nextModule();
    }
    document.getElementById('list').innerHTML=htmlcode;
    setTimeout('refresh()',500);
}
-->
</SCRIPT>
</HEAD>
<BODY onload='refresh();'>
<H1>Device list</H1>
<tt><span id='list'></span></tt>
</BODY>
</HTML>
```

8.4. Error handling

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

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

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

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

As this latest situation is not the most desirable, the Yoctopuce library offers another possibility for error handling, allowing you to create a robust program without needing to catch exceptions at every

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

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

9. Using Yocto-3D 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) server¹, preferably locally on your 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.

9.1. Getting ready

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

- The PHP programming library²
- The VirtualHub software³ 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.

9.2. Control of the Tilt function

A few lines of code are enough to use a Yocto-3D. Here is the skeleton of a PHP code snippet to use the Tilt function.

```
include('yocto_api.php');
include('yocto_tilt.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
$yRegisterHub('http://127.0.0.1:4444/', $errormsg);
$tilt = yFindTilt("Y3DMK001-123456.tilt1");

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

Let's look at these lines in more details.

yocto_api.php and yocto_tilt.php

These two PHP includes provides access to the functions allowing you to manage Yoctopuce modules. `yocto_api.php` must always be included, `yocto_tilt.php` is necessary to manage modules containing a tilt sensor, such as Yocto-3D.

yRegisterHub

The `yRegisterHub` 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.

yFindTilt

The `yFindTilt` function allows you to find a tilt 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-3D module with serial number `Y3DMK001-123456` which you have named "`MyModule`", and for which you have given the `tilt1` function the name "`MyFunction`". The following five calls are strictly equivalent, as long as "`MyFunction`" is defined only once.

```
$tilt = yFindTilt("Y3DMK001-123456.tilt1");
$tilt = yFindTilt("Y3DMK001-123456.MyFunction");
$tilt = yFindTilt("MyModule.tilt1");
$tilt = yFindTilt("MyModule.MyFunction");
$tilt = yFindTilt("MyFunction");
```

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

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `yFindTilt` provides the angle currently measured by the inclinometer. The value returned is a floating number.

yFindCompass and yFindGyro

Functions `yFindCompass`, `yFindGyro`, `yFindMagnetometer` and `yFindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `yFindTilt`.

A real example

Open your preferred text editor⁴, copy the code sample below, save it with the Yoctopuce library files in a location which is accessible to your web server, then use your preferred web browser to access

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

this page. The code is also provided in the directory **Examples/Doc-GettingStarted-Yocto-3D** 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_tilt.php');
    include('yocto_compass.php');
    include('yocto_gyro.php');
    include('yocto_accelerometer.php');

    // Use explicit error handling rather than exceptions
    yDisableExceptions();

    // Setup the API to use the VirtualHub on local machine
    if(yRegisterHub('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
        $anytilt = yFindTilt("$serial.tilt1");
        if (!$anytilt->isOnline()) {
            die("Module not connected (check serial and USB cable)");
        }
    } else {
        // or use any connected module suitable for the demo
        $anytilt = yFirstTilt();
        if(is_null($anytilt)) {
            die("No module connected (check USB cable)");
        } else {
            $serial = $anytilt->module()->get_serialnumber();
        }
    }
    Print("Module to use: <input name='serial' value='$serial'><br>");

    // Get all sensor on the device matching the serial
    $tilt1      = yFindTilt("$serial.tilt1");
    $tilt2      = yFindTilt("$serial.tilt2");
    $compass    = yFindCompass ("$serial.compass");
    $gyro       = yFindGyro ("$serial.gyro");
    $accelerometer = yFindAccelerometer ("$serial.accelerometer");

    $tilt1value     = $tilt1->get_currentValue();
    $tilt2value     = $tilt2 ->get_currentValue();
    $compassvalue   = $compass->get_currentValue();
    $gyrovalue      = $gyro->get_currentValue();
    $accelerometervalue = $accelerometer->get_currentValue();

    Print("tilt1: $tilt1value &deg;<br>");
    Print("tilt2: $tilt2value &deg;<br>");
    Print("compass: $compassvalue &deg;<br>");
    Print("gyro: $gyrovalue &deg;/s<br>");
    Print("Accelerometer: $accelerometervalue g<br>");

    // trigger auto-refresh after one second
    Print("<script language='javascript1.5' type='text/JavaScript'>\n");
    Print("setInterval('window.location.reload()',500);");
    Print("</script>\n");
?>
</BODY>
</HTML>
```

9.3. 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>
<BODY>
<FORM method='get'>
<?php
    include('yocto_api.php');

    // Use explicit error handling rather than exceptions
    yDisableExceptions();

    // Setup the API to use the VirtualHub on local machine
    if(yRegisterHub('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
        $module = yFindModule("$serial");
        if (!$module->isOnline()) {
            die("Module not connected (check serial and USB cable)");
        }
    } else {
        // or use any connected module suitable for the demo
        $module = yFirstModule();
        if($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());
?>
<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
    yDisableExceptions();

    // Setup the API to use the VirtualHub on local machine
    if(yRegisterHub('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
        $module = yFindModule("$serial");
        if (!$module->isOnline()) {
            die("Module not connected (check serial and USB cable)");
        }
    } else {
        // or use any connected module suitable for the demo
        $module = yFirstModule();
        if($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>");
?>
<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 guarantees 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.

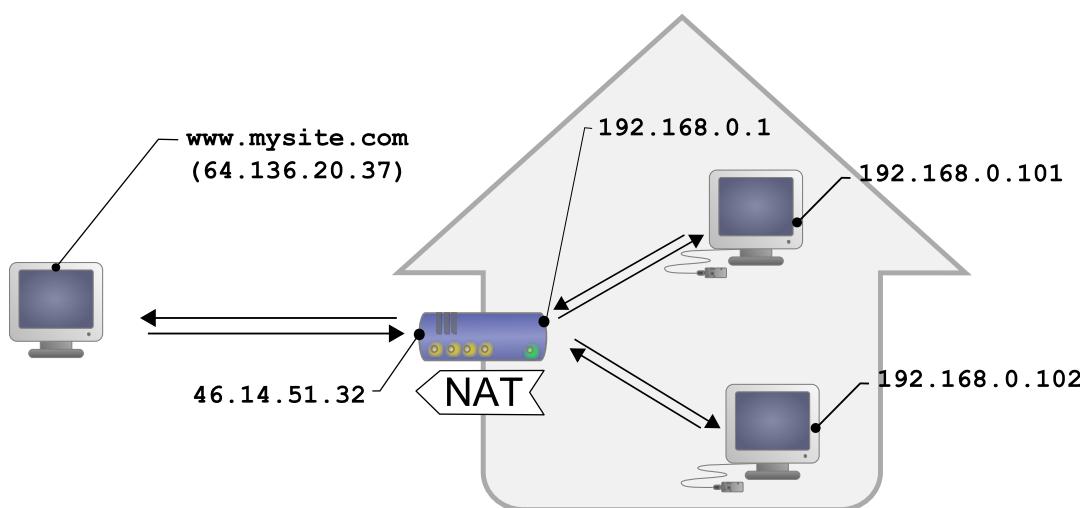
```
<HTML>
<HEAD>
    <TITLE>inventory</TITLE>
</HEAD>
<BODY>
    <H1>Device list</H1>
    <TT>
<?php
    include('yocto_api.php');
    yRegisterHub("http://127.0.0.1:4444/");
    $module = yFirstModule();
    while (!is_null($module)) {
        printf("%s (%s)<br>", $module->get_serialNumber(),
            $module->get_productName());
        $module=$module->nextModule();
    }
?>
</TT>
</BODY>
</HTML>
```

9.4. 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.

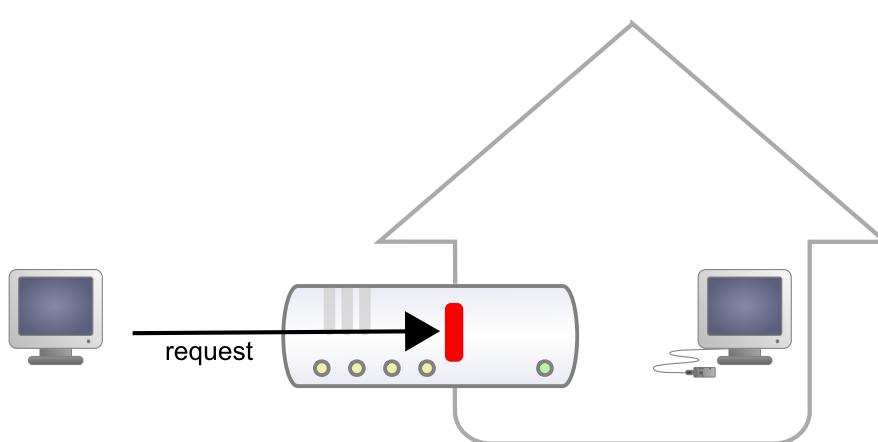
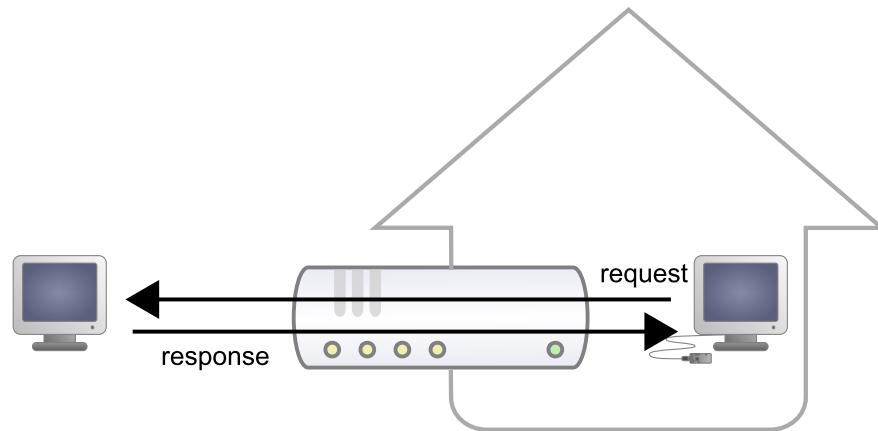
The NAT filter: advantages and disadvantages

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.



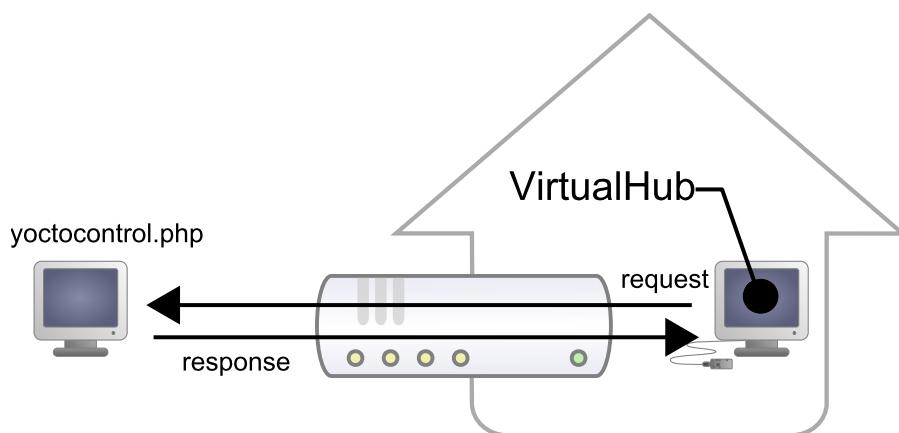
Typical DSL configuration: LAN machines are isolated from the outside by the DSL router

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.



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 at regular intervals.

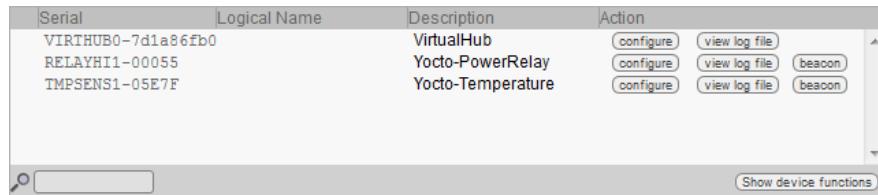


Configuration

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

VIRTHUB0-7d1a86fb09

Edit parameters for VIRTHUB0-7d1a86fb09, and click on the **Save** button.

Serial #	VIRTHUB0-7d1a86fb09
Product name:	VirtualHub
Software version:	10789
Logical name:	<input type="text"/>

Incoming connections

Authentication to read information from the devices: NO

Authentication to make changes to the devices: NO

Outgoing callbacks

Callback URL: octoHub

Delay between callbacks: min: 3 [s] max: 600 [s]

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

Edit callback

This VirtualHub can post the advertised values of all devices on a specific URL on a regular basis. If you wish to use this feature, choose the callback type follow the steps below carefully.

1. Specify the Type of callback you want to use:
2. Specify the URL to use for reporting values. *HTTPS protocol is not yet supported.*
3. If your callback requires authentication, enter credentials here. Digest authentication is recommended, but Basic authentication works as well.
4. Setup the desired frequency of notifications:

No less than seconds between two notification
 But notify after seconds in any case

5. Press on the **Test** button to check your parameters.
6. When everything works, press on the **OK** button.

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 and Node.JS only.

9.5. 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 `yDisableExceptions()` function to commute the library to a mode where exceptions for all the functions are systematically replaced by specific return values, which can be tested by the caller when necessary. For each function, the name of each return value in case of error is systematically documented in the library reference. The name always follows the same logic: a `get_state()` method returns a `Y_STATE_INVALID` value, a `get_currentValue` method returns a `Y_CURRENTVALUE_INVALID` value, and so on. In any case, the returned value is of the expected type and is not a null pointer which would risk crashing your program. At worst, if you display the value without testing it, it will be outside the expected bounds for the returned value. In the case of functions which do not normally return information, the return value is `YAPI_SUCCESS` if everything went well, and a different error code in case of failure.

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

10. Using Yocto-3D with C++

C++ is not the simplest language to master. However, if you take care to limit yourself to its essential functionalities, this language can very well be used for short programs quickly coded, and it has the advantage of being easily ported from one operating system to another. Under Windows, all the examples and the project models are tested with Microsoft Visual Studio 2010 Express, freely available on the Microsoft web site¹. Under Mac OS X, all the examples and project models are tested with XCode 4, available on the App Store. Moreover, under 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++ libraries² are integrally provided as source files. A section of the low-level library is written in pure C, but you should not need to interact directly with it: everything was done to ensure the simplest possible interaction from C++. The library is naturally also available as binary files, so that you can link it directly if you prefer.

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

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

10.1. Control of the Tilt function

A few lines of code are enough to use a Yocto-3D. Here is the skeleton of a C++ code snippet to use the Tilt function.

```
#include "yocto_api.h"
#include "yocto_tilt.h"

[...]
String errmsg;
YTilt *tilt;

// Get access to your device, connected locally on USB for instance
yRegisterHub("usb", errmsg);
tilt = yFindTilt("Y3DMK001-123456.tilt1");
```

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

² www.yoctopuce.com/EN/libraries.php

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

Let's look at these lines in more details.

yocto_api.h et yocto_tilt.h

These two include files provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api.h` must always be used, `yocto_tilt.h` is necessary to manage modules containing a tilt sensor, such as Yocto-3D.

yRegisterHub

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

yFindTilt

The `yFindTilt` function allows you to find a tilt 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-3D module with serial number `Y3DMK001-123456` which you have named "`MyModule`", and for which you have given the `tilt1` function the name "`MyFunction`". The following five calls are strictly equivalent, as long as "`MyFunction`" is defined only once.

```
YTilt *tilt = yFindTilt("Y3DMK001-123456.tilt1");
YTilt *tilt = yFindTilt("Y3DMK001-123456.MyFunction");
YTilt *tilt = yFindTilt("MyModule.tilt1");
YTilt *tilt = yFindTilt("MyModule.MyFunction");
YTilt *tilt = yFindTilt("MyFunction");
```

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

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `yFindTilt` provides the angle currently measured by the inclinometer. The value returned is a floating number.

yFindCompass and yFindGyro

Functions `yFindCompass`, `yFindGyro`, `yFindMagnetometer` and `yFindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `yFindTilt`.

A real example

Launch your C++ environment and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-3D** 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_tilt.h"
```

```

#include "yocto_compass.h"
#include "yocto_gyro.h"
#include "yocto_accelerometer.h"
#include <iostream>
#include <stdlib.h>
#include <iostream>
#include <iomanip>

using namespace std;

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

int main(int argc, const char * argv[])
{
    string errmsg,target;
    YTilt *anytilt,*tilt1, *tilt2;
    YCompass *compass;
    YAccelerometer *accelerometer;
    YGyro *gyro;

    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;
    }

    // try to find a valid serial number
    if(target == "any") {
        anytilt = YTilt::FirstTilt();
        if (anytilt==NULL) {
            cout << "No module connected (check USB cable)" << endl;
            return 1;
        }
    } else {
        anytilt = YTilt::FindTilt(target + ".tilt1");
        if (!anytilt->isOnline()) {
            cout << "Module not connected (check identification and USB cable)" << endl;
            return 1;
        }
    }
    string serial = anytilt->get_module()->get_serialNumber();

    // retrieve all sensors on the device matching the serial
    tilt1 = YTilt::FindTilt(serial + ".tilt1");
    tilt2 = YTilt::FindTilt(serial + ".tilt2");
    compass = YCompass::FindCompass(serial + ".compass");
    accelerometer = YAccelerometer::FindAccelerometer(serial + ".accelerometer");
    gyro = YGyro::FindGyro(serial + ".gyro");
    int count = 0;

    while(1) {
        if(!tilt1->isOnline()) {
            cout << "device disconnected";
            break;
        }
        if ((count % 10) == 0) {
            cout << "tilt1\n" << tilt2->compass->tacc->gyro" << endl;
        }
        cout << std::setprecision(2) << std::setw(8) << tilt1->get_currentValue() << "\t"
            << tilt2->get_currentValue() << "\t"
            << compass->get_currentValue() << "\t"
            << accelerometer->get_currentValue() << "\t"
            << gyro->get_currentValue() << endl;
        count++;
    }
}

```

```

        count++;
        YAPI::Sleep(250,errmsg);
    }

    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.

```

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

#include "yocto_api.h"

using namespace std;

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

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

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

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

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

    if (module->isOnline()) {
        if (argc > 2) {
            if (string(argv[2]) == "ON")
                module->setBeacon(Y_BEACON_ON);
            else
                module->setBeacon(Y_BEACON_OFF);
        }
        cout << "serial: " << module->getSerialNumber() << endl;
        cout << "logical name: " << module->getLogicalName() << endl;
        cout << "luminosity: " << module->getLuminosity() << endl;
        cout << "beacon: ";
        if (module->getBeacon() == Y_BEACON_ON)
            cout << "ON" << endl;
        else
            cout << "OFF" << endl;
        cout << "upTime: " << module->getUpTime()/1000 << " sec" << endl;
        cout << "USB current: " << module->getUsbCurrent() << " mA" << endl;
        cout << "Logs:" << endl << module->getLastLogs() << endl;
    } else {
        cout << argv[1] << " not connected (check identification and USB cable)" << endl;
    }
    return 0;
}

```

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

Changing the module settings

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

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

#include "yocto_api.h"

using namespace std;

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

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

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

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

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

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

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

Listing the modules

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

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

```

using namespace std;

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

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

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

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

```

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

When you work without exceptions, you can obtain an error code and an error message explaining the source of the error. You can request them from the object which returned the error, calling the

`errType()` and `errMessage()` methods. Their returned values contain the same information as in the exceptions when they are active.

10.4. Integration variants for the C++ Yoctopuce library

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

Integration in source format

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

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

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

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

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

Integration as a static library

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

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

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

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

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

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

Integration as a dynamic library

Integration of the Yoctopuce library as a dynamic library allows you to produce an executable smaller than with the two previous methods, and to possibly update this library, if a patch reveals itself necessary, without needing to recompile the source code of the application. On the other hand, it is an integration mode which systematically requires you to copy the dynamic library on the target

machine where the application will run (**yocto.dll** for Windows, **libyocto.so.1.0.1** for Mac OS X and Linux).

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

Then, for your 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**, **lpthread**, **libusb-1.0**, and **libstdc++**.

With GCC, the command line to compile is simply:

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

11. Using Yocto-3D 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 libraries¹ 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 example² with video shots showing how to integrate the library into your projects.

11.1. Control of the Tilt function

Launch Xcode 4.2 and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-3D** of the Yoctopuce library.

```
#import <Foundation/Foundation.h>
#import "yocto_api.h"
#import "yocto_tilt.h"
#import "yocto_compass.h"
#import "yocto_gyro.h"
#import "yocto_accelerometer.h"

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

¹ www.yoctopuce.com/EN/libraries.php

² www.yoctopuce.com/EN/article/new-objective-c-library-for-mac-os-x

```

    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(@"%@", [error localizedDescription]);
            return 1;
        }

        NSString *target = [NSString stringWithUTF8String:argv[1]];
        YTilt *anytilt,*tilt1, *tilt2;
        YCompass *compass;
        YAcelerometer *accelerometer;
        YGyro *gyro;

        if([target isEqualToString:@"any"]){
            anytilt = [YTilt FirstTilt];
            if (anytilt==NULL) {
                NSLog(@"No module connected (check USB cable)");
                return 1;
            }
        } else {
            anytilt = [YTilt FindTilt:[target stringByAppendingPathComponent:@".tilt1"]];
            if(![anytilt isOnline]) {
                NSLog(@"Module not connected (check identification and USB cable)");
                return 1;
            }
        }
        NSString *serial = [[anytilt get_module] get_serialNumber];
        // retrieve all sensors on the device matching the serial
        tilt1 = [YTilt FindTilt:[serial stringByAppendingPathComponent:@".tilt1"]];
        tilt2 = [YTilt FindTilt:[serial stringByAppendingPathComponent:@".tilt2"]];
        compass = [YCompass FindCompass:[serial stringByAppendingPathComponent:@".compass"]];
        accelerometer = [YAcelerometer FindAccelerometer:[serial
stringByAppendingString:@".accelerometer"]];
        gyro = [YGyro FindGyro:[serial stringByAppendingPathComponent:@".gyro"]];
        int count = 0;

        while(1) {
            if(![tilt1 isOnline]) {
                NSLog(@"device disconnected");
                break;
            }
            if ((count % 10) == 0)
                NSLog(@"%@", [tilt1 get_currentValue],
                [tilt2 get_currentValue],
                [compass get_currentValue],
                [accelerometer get_currentValue],
                [gyro get_currentValue]);
            count++;

            [YAPI Sleep:250:NULL];
        }
        [YAPI FreeAPI];
    }

    return 0;
}

```

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

yocto_api.h et yocto_tilt.h

These two import files provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api.h` must always be used, `yocto_tilt.h` is necessary to manage modules containing a tilt sensor, such as Yocto-3D.

[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.

[Tilt FindTilt]

The `[Tilt FindTilt]` function allows you to find a tilt 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-3D module with serial number `Y3DMK001-123456` which you have named "`MyModule`", and for which you have given the `tilt1` function the name "`MyFunction`". The following five calls are strictly equivalent, as long as "`MyFunction`" is defined only once.

```
YTilt *tilt = [Tilt FindTilt:@"Y3DMK001-123456.tilt1"];
YTilt *tilt = [Tilt FindTilt:@"Y3DMK001-123456.MyFunction"];
YTilt *tilt = [Tilt FindTilt:@"MyModule.tilt1"];
YTilt *tilt = [Tilt FindTilt:@"MyModule.MyFunction"];
YTilt *tilt = [Tilt FindTilt:@"MyFunction"];
```

`[Tilt FindTilt]` returns an object which you can then use at will to control the tilt sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `YTilt.FindTilt` provides the value of orientation measured by the tilt sensor. The value returned is a floating number.

YCompass.FindCompass, YGyro.FindGyro...

Functions `YCompass.FindCompass`, `YMagnetometer.FindMagnetometer`, `YGyro.FindGyro` and `YAccelerometer.FindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `YTilt.FindTilt`.

11.2. 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: %@", [module serialNumber]);
        NSLog(@"logical name: %@", [module logicalName]);
        NSLog(@"luminosity: %d", [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: %@", [module get_lastLogs]);
    } else {
        NSLog(@"%@", not connected (check identification and USB cable)\n",
               serial_or_name);
    }
}
return 0;
}

```

Each property `xxx` of the module can be read thanks to a method of type `get_xxx`, 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: %@", module.logicalName);
} else {
    NSLog(@"%@", module.logicalName);
}
}
return 0;
}

```

Warning: the number of write cycles of the nonvolatile memory of the module is limited. When this limit is reached, nothing guarantees 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: %@", [error localizedDescription]);
            return 1;
        }

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

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

```

11.3. Error handling

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

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

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

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

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

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

12. Using Yocto-3D 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 site¹.

12.1. Installation

Download the Visual Basic Yoctopuce library from the Yoctopuce web site². 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.

12.2. 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 modules³. The source files in Visual Basic manage the high level part of the API. Therefore, you 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` directory⁴. 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 Tilt function

A few lines of code are enough to use a Yocto-3D. Here is the skeleton of a Visual Basic code snippet to use the Tilt function.

```
[...]
Dim errmsg As String errmsg
Dim tilt As YTilt

REM Get access to your device, connected locally on USB for instance
yRegisterHub("usb", errmsg)
tilt = yFindTilt("Y3DMK001-123456.tilt1")

REM Hot-plug is easy: just check that the device is online
If (tilt.isOnline()) Then
    REM Use tilt.get_currentValue(), ...
End If
```

Let's look at these lines in more details.

yRegisterHub

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

yFindTilt

The `yFindTilt` function allows you to find a tilt 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-3D module with serial number `Y3DMK001-123456` which you have named "`MyModule`", and for which you have given the `tilt1` function the name "`MyFunction`". The following five calls are strictly equivalent, as long as "`MyFunction`" is defined only once.

```
tilt = yFindTilt("Y3DMK001-123456.tilt1")
tilt = yFindTilt("Y3DMK001-123456.MyFunction")
tilt = yFindTilt("MyModule.tilt1")
tilt = yFindTilt("MyModule.MyFunction")
tilt = yFindTilt("MyFunction")
```

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

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `yFindTilt` provides the angle currently measured by the inclinometer. The value returned is a floating number.

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

yFindCompass and yFindGyro

Functions `yFindCompass`, `yFindGyro`, `yFindMagnetometer` and `yFindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `YFindTilt`.

A real example

Launch Microsoft VisualBasic and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-3D** 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 serial As String
    Dim count As Integer

    Dim anytilt, tilt1, tilt2 As YTilt
    Dim compass As YCompass
    Dim accelerometer As YAccelerometer
    Dim gyro As YGyro

    If argv.Length < 2 Then Usage()

    target = argv(1)

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

    If target = "any" Then
        anytilt = yFirstTilt()
        If anytilt Is Nothing Then
            Console.WriteLine("No module connected (check USB cable)")
            End
        End If
    Else
        anytilt = yFindTilt(target + ".tilt1")
        If Not (anytilt.isOnline()) Then
            Console.WriteLine("Module not connected (check identification and USB cable)")
            End
        End If
    End If

    serial = anytilt.get_module().get_serialNumber()
    tilt1 = YTilt.FindTilt(serial + ".tilt1")
    tilt2 = YTilt.FindTilt(serial + ".tilt2")
    compass = YCompass.FindCompass(serial + ".compass")
    accelerometer = YAccelerometer.FindAccelerometer(serial + ".accelerometer")
    gyro = YGyro.FindGyro(serial + ".gyro")
    count = 0

    While (True)
        If (Not tilt1.isOnline()) Then
            Console.WriteLine("Module disconnected")
        End If
    End While
End Sub
```

```

        End
        End If

        If (count Mod 10 = 0) Then
            Console.WriteLine("tilt1" + Chr(9) + "tilt2" + Chr(9) + "compass" + Chr(9) + "acc"
+ Chr(9) + "gyro")
        End If

        Console.Write(tilt1.get_currentValue().ToString() + Chr(9))
        Console.Write(tilt2.get_currentValue().ToString() + Chr(9))
        Console.Write(compass.get_currentValue().ToString() + Chr(9))
        Console.Write(accelerometer.get_currentValue().ToString() + Chr(9))
        Console.WriteLine(gyro.get_currentValue().ToString())
        count = count + 1
        ySleep(250, errmsg)

    End While

End Sub

End Module

```

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.

```

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

        If argv.Length < 2 Then usage()

        m = yFindModule(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.WriteLine("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
    End Sub

```

```

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

    m = yFindModule(argv(1)) REM use serial or logical name
    If m.isOnline() Then

        newname = argv(2)
        If (Not yCheckLogicalName(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

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

    Console.WriteLine("Device list")
    M = yFirstModule()
    While M IsNot Nothing
        Console.WriteLine(M.get_serialNumber() + " (" + M.get_productName() + ")")
        M = M.nextModule()
    End While

End Sub

End Module
```

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 `yDisableExceptions()` function to commute the library to a mode where exceptions for all the functions are systematically replaced by specific return values, which can be tested by the caller when necessary. For each function, the name of each return value in case of error is systematically documented in the library reference. The name always follows the same logic: a `get_state()` method returns a `Y_STATE_INVALID` value, a `get_currentValue` method returns a `Y_CURRENTVALUE_INVALID` value, and so on. In any case, the returned value is of the expected type and is not a null pointer which would risk crashing

your program. At worst, if you display the value without testing it, it will be outside the expected bounds for the returned value. In the case of functions which do not normally return information, the return value is `YAPI_SUCCESS` if everything went well, and a different error code in case of failure.

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

13. Using Yocto-3D 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 site¹.

13.1. Installation

Download the Visual C# Yoctopuce library from the Yoctopuce web site². 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.

13.2. 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 modules³. The source files in Visual C# manage the high level part of the API. Therefore, you 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.

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-csharp-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` directory⁴. 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.

13.3. Control of the Tilt function

A few lines of code are enough to use a Yocto-3D. Here is the skeleton of a C# code snippet to use the Tilt function.

```
[...]
string errmsg = "";
YTilt tilt;

// Get access to your device, connected locally on USB for instance
YAPI.RegisterHub("usb", errmsg);
tilt = YTilt.FindTilt("Y3DMK001-123456.tilt1");

// Hot-plug is easy: just check that the device is online
if (tilt.isOnline())
{
    // Use tilt.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.

YTilt.FindTilt

The `YTilt.FindTilt` function allows you to find a tilt 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-3D module with serial number `Y3DMK001-123456` which you have named "`MyModule`", and for which you have given the `tilt1` function the name "`MyFunction`". The following five calls are strictly equivalent, as long as "`MyFunction`" is defined only once.

```
tilt = YTilt.FindTilt("Y3DMK001-123456.tilt1");
tilt = YTilt.FindTilt("Y3DMK001-123456.MyFunction");
tilt = YTilt.FindTilt("MyModule.tilt1");
tilt = YTilt.FindTilt("MyModule.MyFunction");
tilt = YTilt.FindTilt("MyFunction");
```

`YTilt.FindTilt` returns an object which you can then use at will to control the tilt sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `YTilt.FindTilt` provides the value of orientation measured by the tilt sensor. The value returned is a floating number.

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

YCompass.FindCompass, YGyro.FindGyro...

Functions `YCompass.FindCompass`, `YMagnometer.FindMagnometer`, `YGyro.FindGyro` and `YAccelerometer.FindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `YTilt.FindTilt`.

A real example

Launch Microsoft Visual C# and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-3D** 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(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;

            YTilt anytilt, tilt1, tilt2;
            YCompass compass;
            YAccelerometer accelerometer;
            YGyro gyro;

            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")
            {
                anytilt = YTilt.FirstTilt();
                if (anytilt == null)
                {
                    Console.WriteLine("No module connected (check USB cable)");
                    Environment.Exit(0);
                }
            }
            else
            {
                anytilt = YTilt.FindTilt(target + ".tilt1");
                if (!anytilt.isOnline())
                {
                    Console.WriteLine("Module not connected (check identification and USB
cable)");
                    Environment.Exit(0);
                }
            }

            string serial = anytilt.get_module().get_serialNumber();
            tilt1 = YTilt.FindTilt(serial + ".tilt1");
        }
    }
}
```

```
tilt2 = YTilt.FindTilt(serial + ".tilt2");
compass = YCompass.FindCompass(serial + ".compass");
accelerometer = YAccelerometer.FindAccelerometer(serial + ".accelerometer");
gyro = YGyro.FindGyro(serial + ".gyro");
int count = 0;

while (true)
{
    if (!tilt1.isOnline())
    {
        Console.WriteLine("device disconnected");
        Environment.Exit(0);
    }

    if (count % 10 == 0) Console.WriteLine("tilt1    tilt2    compass    acc
gyro");

    Console.WriteLine(tilt1.get_currentValue().ToString() + "\t");
    Console.WriteLine(tilt2.get_currentValue().ToString() + "\t");
    Console.WriteLine(compass.get_currentValue().ToString() + "\t");
    Console.WriteLine(accelerometer.get_currentValue().ToString() + "\t");
    Console.WriteLine(gyro.get_currentValue().ToString());

    YAPI.Sleep(250, ref errmsg);
}

}

}

}
```

13.4. 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)");
}

}
}
}

```

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");
}
}
}
```

Warning: the number of write cycles of the nonvolatile memory of the module is limited. When this limit is reached, nothing guarantees 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();
            }
        }
    }
}
```

13.5. 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 `yDisableExceptions()` function to commute the library to a mode where exceptions for all the functions are systematically replaced by specific return values, which can be tested by the caller when necessary. For each function, the name of each return value in case of error is systematically documented in the library reference. The name always follows the same logic: a `get_state()` method returns a `Y_STATE_INVALID` value, a `get_currentValue` method returns a `Y_CURRENTVALUE_INVALID` value, and so on. In any case, the returned value is of the expected type and is not a null pointer which would risk crashing your program. At worst, if you display the value without testing it, it will be outside the expected bounds for the returned value. In the case of functions which do not normally return information, the return value is `YAPI_SUCCESS` if everything went well, and a different error code in case of failure.

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

14. Using Yocto-3D with Delphi

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

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

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

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

14.1. Preparation

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

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

14.2. Control of the Tilt function

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

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

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.

```

yocto_compass,
yocto_accelerometer,
yocto_gyro;

Procedure Usage();
var
  exe : string;

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

var
  m           : TYmodule;
  anytilt,tilt1,tilt2   : TYTilt;
  Compass      : TYcompass;
  accelerometer : TYAccelerometer;
  gyro         : TYGyro;
  errmsg,serial : string;
  done         : boolean;
  count        : integer;

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
    // lets try to find the first available tilt sensor
    anytilt := yFirstTilt();
    if anytilt=nil then
    begin
      writeln('No module connected (check USB cable)');
      halt;
    end
  end
  else
  // or the one specified on command line
  anytilt:= YFindTilt(paramstr(1)+'.tilt1');

  // make sure it is online
  if not anytilt.isOnline() then
  begin
    writeln('No module connected (check USB cable)');
    halt;
  end;

  // lets find the parent module so we can get the other sensors
  m     := anytilt.get_module();
  serial := m.get_serialNumber();

  // retreive some sensors present on the yocto-3D
  tilt1      := yFindTilt(serial+'.tilt1');
  tilt2      := yFindTilt(serial+'.tilt2');
  compass    := yFindCompass(serial+'.compass');
  accelerometer := yFindaccelerometer(serial+'.accelerometer');
  gyro       :=yFindGyro(serial+'.gyro');

  // let's poll
  done := false;
  count :=0;

  repeat
    if (tilt1.isOnline()) then
    begin
      if (count mod 10=0) then  Writeln('tilt1'#9'tilt2'#9'compass'#9'acc'#9'gyro');
      Write(FloatToStr(tilt1.get_currentValue())+#9);
      Write(FloatToStr(tilt2.get_currentValue())+#9);
    end;
    count := count+1;
  until done;
end;

```

```

        Write(FloatToStr(compass.get_currentValue())+#9);
        Write(FloatToStr(accelerometer.get_currentValue())+#9);
        Writeln(FloatToStr(gyro.get_currentValue()));
        inc(count);
        Sleep(100);
    end
else
begin
    Writeln('Module not connected (check identification and USB cable)');
    done := true;
end;
until done;
end.
```

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

yocto_api and yocto_tilt

These two units provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api` must always be used, `yocto_tilt` is necessary to manage modules containing a tilt sensor, such as Yocto-3D.

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.

yFindTilt

The `yFindTilt` function allows you to find a tilt 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-3D module with serial number `Y3DMK001-123456` which you have named "`MyModule`", and for which you have given the `tilt1` function the name "`MyFunction`". The following five calls are strictly equivalent, as long as "`MyFunction`" is defined only once.

```

tilt := yFindTilt("Y3DMK001-123456.tilt1");
tilt := yFindTilt("Y3DMK001-123456.MyFunction");
tilt := yFindTilt("MyModule.tilt1");
tilt := yFindTilt("MyModule.MyFunction");
tilt := yFindTilt("MyFunction");
```

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

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `yFindTilt` provides the angle currently measured by the inclinometer. The value returned is a floating number.

yFindCompass and yFindGyro

Functions `yFindCompass`, `yFindGyro`, `yFindMagnetometer` and `yFindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `yFindTilt`.

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 = 'Y3DMK001-123456'; // use serial number or logical name

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

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

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

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

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

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

```

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

Changing the module settings

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

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

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

const
  serial = 'Y3DMK001-123456'; // 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
    Writeln('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();

  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 guarantees that the saving process is performed correctly. This limit, linked to the technology employed by the module micro-processor, is located at about 100000 cycles. In short, you can use the `saveToFlash()` function only 100000 times in the life of the module. Make sure you do not call this function within a loop.

Listing the modules

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

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

var
  module : TYModule;
  errmsg : string;

begin
  // Setup the API to use local USB devices
  if yRegisterHub('usb', errmsg)<>YAPI_SUCCESS then
  begin
```

```

Write('RegisterHub error: '+errmsg);
exit;
end;

Writeln('Device list');

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

end.

```

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

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

15. Using the Yocto-3D 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 site¹.

15.1. Source files

The Yoctopuce library classes² 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 Tilt function

A few lines of code are enough to use a Yocto-3D. Here is the skeleton of a Python code snippet to use the Tilt function.

¹ <http://www.python.org/download/>

² www.yoctopuce.com/EN/libraries.php

```
[...]
errmsg=YRefParam()
#Get access to your device, connected locally on USB for instance
YAPI.RegisterHub("usb",errmsg)
tilt = YTilt.FindTilt("Y3DMK001-123456.tilt1")

# Hot-plug is easy: just check that the device is online
if tilt.isOnline():
    #Use tilt.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.

YTilt.FindTilt

The `YTilt.FindTilt` function allows you to find a tilt 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-3D module with serial number `Y3DMK001-123456` which you have named "`MyModule`", and for which you have given the `tilt1` function the name "`MyFunction`". The following five calls are strictly equivalent, as long as "`MyFunction`" is defined only once.

```
tilt = YTilt.FindTilt("Y3DMK001-123456.tilt1")
tilt = YTilt.FindTilt("Y3DMK001-123456.MyFunction")
tilt = YTilt.FindTilt("MyModule.tilt1")
tilt = YTilt.FindTilt("MyModule.MyFunction")
tilt = YTilt.FindTilt("MyFunction")
```

`YTilt.FindTilt` returns an object which you can then use at will to control the tilt sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `YTilt.FindTilt` provides the value of orientation measured by the tilt sensor. The value returned is a floating number.

YCompass.FindCompass, YGyro.FindGyro...

Functions `YCompass.FindCompass`, `YMagnetometer.FindMagnetometer`, `YGyro.FindGyro` and `YAccelerometer.FindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `YTilt.FindTilt`.

A real example

Launch Python and open the corresponding sample script provided in the directory **Examples/Doc-GettingStarted-Yocto-3D** 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_tilt import *
from yocto_compass import *
from yocto_gyro import *
from yocto_accelerometer 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)')

errormsg=YRefParam()

if len(sys.argv)<2 : usage()

target=sys.argv[1]

# Setup the API to use local USB devices
if YAPI.RegisterHub("usb", errormsg)!= YAPI.SUCCESS:
    sys.exit("init error"+errormsg.value)

if target=='any':
    # retreive any tilt sensor
    anytilt = YTilt.FirstTilt()
    if anytilt is None :
        die('No module connected (check USB cable)')
    m = anytilt.get_module()
    target = m.get_serialNumber()
else:
    anytilt = YTilt.FindTilt(target + ".tilt1")
    if not (anytilt.isOnline()):
        die('Module not connected (check identification and USB cable)')

serial = anytilt.get_module().get_serialNumber()
tilt1 = YTilt.FindTilt(serial + ".tilt1")
tilt2 = YTilt.FindTilt(serial + ".tilt2")
compass = YCompass.FindCompass(serial + ".compass")
accelerometer = YAccelerometer.FindAccelerometer(serial+".accelerometer")
gyro = YGyro.FindGyro(serial + ".gyro")

count =0

while (True):
    if not(tilt1.isOnline()):
        die("Module not connected (check identification and USB cable)")

    if (count % 10 == 0): print("tilt1   tilt2   compass acc      gyro")

    print(  "%-7.1f "%tilt1.get_currentValue() + \
            "%-7.1f "%tilt2.get_currentValue() + \
            "%-7.1f "%compass.get_currentValue() + \
            "%-7.1f "%accelerometer.get_currentValue() + \
            "%-7.1f"%gyro.get_currentValue())
    count=count+1
    YAPI.Sleep(250, errormsg)

```

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)")

```

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")

```

Warning: the number of write cycles of the nonvolatile memory of the module is limited. When this limit is reached, nothing guarantees 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()
```

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 `yDisableExceptions()` function to commute the library to a mode where exceptions for all the functions are systematically replaced by specific return values, which can be tested by the caller when necessary. For each function, the name of each return value in case of error is systematically documented in the library reference. The name always

follows the same logic: a `get_state()` method returns a `Y_STATE_INVALID` value, a `get_currentValue` method returns a `Y_CURRENTVALUE_INVALID` value, and so on. In any case, the returned value is of the expected type and is not a null pointer which would risk crashing your program. At worst, if you display the value without testing it, it will be outside the expected bounds for the returned value. In the case of functions which do not normally return information, the return value is `YAPI_SUCCESS` if everything went well, and a different error code in case of failure.

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

16. Using the Yocto-3D 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 library¹
- The VirtualHub software² 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 Tilt function

A few lines of code are enough to use a Yocto-3D. Here is the skeleton of a Java code snippet to use the Tilt function.

```
[...]  
  
// Get access to your device, connected locally on USB for instance  
YAPI.RegisterHub("127.0.0.1");  
tilt = YTilt.FindTilt("Y3DMK001-123456.tilt1");  
  
// Hot-plug is easy: just check that the device is online  
if (tilt.isOnline())  
{ //Use tilt.getCurrentValue()  
  ...  
}
```

¹ www.yoctopuce.com/EN/libraries.php

² www.yoctopuce.com/EN/virtualhub.php

[...]

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.

YTilt.FindTilt

The `YTilt.FindTilt` function allows you to find a tilt 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-3D module with serial number `Y3DMK001-123456` which you have named "`MyModule`", and for which you have given the `tilt1` function the name "`MyFunction`". The following five calls are strictly equivalent, as long as "`MyFunction`" is defined only once.

```
tilt = YTilt.FindTilt("Y3DMK001-123456.tilt1")
tilt = YTilt.FindTilt("Y3DMK001-123456.MyFunction")
tilt = YTilt.FindTilt("MyModule.tilt1")
tilt = YTilt.FindTilt("MyModule.MyFunction")
tilt = YTilt.FindTilt("MyFunction")
```

`YTilt.FindTilt` returns an object which you can then use at will to control the tilt sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `YTilt.FindTilt` provides the value of orientation measured by the tilt sensor. The value returned is a floating number.

YCompass.FindCompass, YGyro.FindGyro...

Functions `YCompass.FindCompass`, `YMagnetometer.FindMagnetometer`, `YGyro.FindGyro` and `YAccelerometer.FindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `YTilt.FindTilt`.

A real example

Launch your Java environment and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-3D** 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);
        }
    }
}
```

```

YTilt anytilt,tilt1, tilt2;
YCompass compass;
YAccelerometer accelerometer;
YGyro gyro;

if (args.length == 0) {
    anytilt = YTilt.FirstTilt();
    if (anytilt == null) {
        System.out.println("No module connected (check USB cable)");
        System.exit(1);
    }
} else {
    anytilt = YTilt.FindTilt(args[0] + ".tilt1");
    if (!anytilt.isOnline()){
        System.out.println("Module not connected (check identification and USB
cable)");
        System.exit(1);
    }
}
try {
    String serial = anytilt.get_module().get_serialNumber();
    tilt1 = YTilt.FindTilt(serial + ".tilt1");
    tilt2 = YTilt.FindTilt(serial + ".tilt2");
    compass = YCompass.FindCompass(serial + ".compass");
    accelerometer = YAccelerometer.FindAccelerometer(serial + ".accelerometer");
    gyro = YGyro.FindGyro(serial + ".gyro");
    int count = 0;
    while (true) {
        if (!tilt1.isOnline()) {
            System.out.println("device disconnected");
            System.exit(0);
        }

        if (count % 10 == 0)
            System.out.println("tilt1   tilt2   compass   acc   gyro");

        System.out.println("") + tilt1.get_currentValue() + "\t" +
tilt2.get_currentValue() +
"\t" + compass.get_currentValue() + "\t" +
accelerometer.get_currentValue() + "\t" +
gyro.get_currentValue());
        count++;
        YAPI.Sleep(250);
    }
} catch (YAPI_Exception ex) {
    System.out.println("Module not connected (check identification and USB cable)");
};
}
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 guarantees 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-3D 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 library¹. 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, you 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 Transformer 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 YoctoHub².

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, you 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 Tilt function

A few lines of code are enough to use a Yocto-3D. Here is the skeleton of a Java code snippet to use the Tilt function.

```
[...]
// Retrieving the object representing the module (connected here locally by USB)
YAPI.EnableUSBHost(this);
YAPI.RegisterHub("usb");
tilt = YTilt.FindTilt("Y3DMK001-123456.tilt1");

// Hot-plug is easy: just check that the device is online
if (tilt.isOnline())
    { //Use tilt.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.

YTilt.FindTilt

The `YTilt.FindTilt` function allows you to find a tilt 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-3D module with serial number `Y3DMK001-123456` which you have named "`MyModule`", and for which you have given the `tilt1` function the name "`MyFunction`". The following five calls are strictly equivalent, as long as "`MyFunction`" is defined only once.

```
tilt = YTilt.FindTilt("Y3DMK001-123456.tilt1")
tilt = YTilt.FindTilt("Y3DMK001-123456.MyFunction")
tilt = YTilt.FindTilt("MyModule.tilt1")
tilt = YTilt.FindTilt("MyModule.MyFunction")
tilt = YTilt.FindTilt("MyFunction")
```

`YTilt.FindTilt` returns an object which you can then use at will to control the tilt sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `YTilt.FindTilt` provides the value of orientation measured by the tilt sensor. The value returned is a floating number.

YCompass.FindCompass, YGyro.FindGyro...

Functions `YCompass.FindCompass`, `YMagnetometer.FindMagneter`, `YGyro.FindGyro` and `YAccelerometer.FindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `YTilt.FindTilt`.

A real example

Launch your 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.YAccelerometer;
import com.yoctopuce.YoctoAPI.YCompass;
import com.yoctopuce.YoctoAPI.YGyro;
import com.yoctopuce.YoctoAPI.YModule;
import com.yoctopuce.YoctoAPI.YSensor;
import com.yoctopuce.YoctoAPI.YTilt;

public class GettingStarted_Yocto_3D 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_meteo);
        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.getProduct().equals("Yocto-3D")) {
                    String serial = module.getSerialNumber();
                    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) {
                YSensor tilt1 = YTilt.FindTilt(serial + ".tilt1");
                try {
                    TextView view = (TextView) findViewById(R.id.tilt1field);
                    view.setText(String.format("%.1f %s", tilt1.getCurrentValue(),
tilt1.getUnit()));
                } catch (YAPI_Exception e) {
                    e.printStackTrace();
                }
                YTilt tilt2 = YTilt.FindTilt(serial + ".tilt2");
                try {
                    TextView view = (TextView) findViewById(R.id.tilt2field);
                    view.setText(String.format("%.1f %s", tilt2.getCurrentValue(),
tilt2.getUnit()));
                } catch (YAPI_Exception e) {
                    e.printStackTrace();
                }
                YCompass compass = YCompass.FindCompass(serial + ".compass");
                try {
                    TextView view = (TextView) findViewById(R.id.compassfield);
                    view.setText(String.format("%.1f %s", compass.getCurrentValue(),
compass.getUnit()));
                } catch (YAPI_Exception e) {
                    e.printStackTrace();
                }
                YAccelerometer accelerometer = YAccelerometer.FindAccelerometer(serial +
".accelerometer");
                try {
                    TextView view = (TextView) findViewById(R.id.accelfield);
                    view.setText(String.format("%.1f %s", accelerometer.getCurrentValue(),
accelerometer.getUnit()));
                } catch (YAPI_Exception e) {
                    e.printStackTrace();
                }
                YGyro gyro = YGyro.FindGyro(serial + ".gyro");
                try {
                    TextView view = (TextView) findViewById(R.id.gyrofield);
                    view.setText(String.format("%.1f %s", gyro.getCurrentValue(),
gyro.getUnit()));
                } catch (YAPI_Exception e) {
                    e.printStackTrace();
                }
            }
            handler.postDelayed(this, 200);
        }
    };
}

```

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.util.Log;
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);
        Log.d("switch", "beacon" + module.getBeacon());
        sw.setChecked(module.getBeacon() == YModule.BEACON_ON);
        field = (TextView) findViewById(R.id.logs);
        field.setText(module.getLastLogs());

    } 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 `xxxx` 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.getHardwareId();
                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 guarantees 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.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);
            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 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.

18.1. 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.

18.2. 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-3D with the `Y3DMK001-12345` 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/Y3DMK001-12345/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/Y3DMK001-12345/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/Y3DMK001-12345/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 `tilt1` function, build the following URL:

```
http://127.0.0.1:4444/bySerial/Y3DMK001-12345/api/tilt1.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/Y3DMK001-12345/api/tilt1/logicalName
```

Rather logically, attributes can be modified in the same manner.

```
http://127.0.0.1:4444/bySerial/Y3DMK001-12345/api/tilt1?logicalName=myFunction
```

You can find the list of available attributes for your Yocto-3D 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/Y3DMK001-12345/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/Y3DMK001-12345/dataLogger.json?id=tilt1&utc=1389801080
```

18.3. 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
libyapi-armhf.so	Linux ARM HL
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 bufsize, 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. *bufsize* 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 result 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;
                         bufsize: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        = 'Y3DMK001-12345';
  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 yapi GetAllDevices(int *buffer,int maxsize,int *neededsize,char *errmsg);
int yapi GetDeviceInfo(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 *yapi GetAllDevices*. *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 yapi GetAllDevices(@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_*.

18.4. 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.

19. Advanced programming

The preceding chapters have introduced, in each available language, the basic programming functions which can be used with your Yocto-3D 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.

19.1. 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 classic 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 magnetometer changes in a significant way. It takes as parameter the relevant function and the new value, as a character string.¹

```
static void valueChangeCallback(YMagnetometer fct, string value)
{
    Console.WriteLine(fct.getHardwareId() + "=" + 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 YMagnetometer 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 Magnetometer function with the help of the `registerValueCallback` method. The following example sets up a callback for the first available Magnetometer function.

```
YMagnetometer f = YMagnetometer.FirstMagnetometer();
f.registerValueCallback(magnetometerChangeCallBack)
```

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(YMagnetometer fct, YMeasure measure)
{
    Console.WriteLine(fct.getHardwareId() + "=" +
                      measure.getAverageValue());
}
```

Setting up a timed report callback

The callback is set up for a given Magnetometer function with the help of the `registerTimedReportCallback` method. The callback will only be invoked once a callback frequency has been set using `setReportFrequency` (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 the first available Magnetometer function.

¹ The value passed as parameter is the same as the value returned by the `getAdvertisedValue()` method.

```
YMagnetometer f = YMagnetometer.FirstMagnetometer();
f.set_reportFrequency("4/m");
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 `YMagnetometer`. Thus, the same callback function will be usable with any subclass of `YSensor` (and in particular with `YMagnetometer`). With the callback function, you can use the method `get_unit()` 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.

19.2. The data logger

Your Yocto-3D 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.

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-3D 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-3D 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-3D 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 an 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 detailed measures (that grows while `loadMore` is being called repeatedly)

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-3D does not have a battery, it cannot guess alone the current time when powered on. Nevertheless, the Yocto-3D 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-3D 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-3D 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-3D 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-3D 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-3D 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-3D 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-3D might be subject to a clock skew (theoretically up to 0.3%).

19.3. Sensor calibration

Your Yocto-3D 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-3D. 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-3D 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-3D. On the other hand, the API supports an external arbitrary calibration, implemented with callbacks.

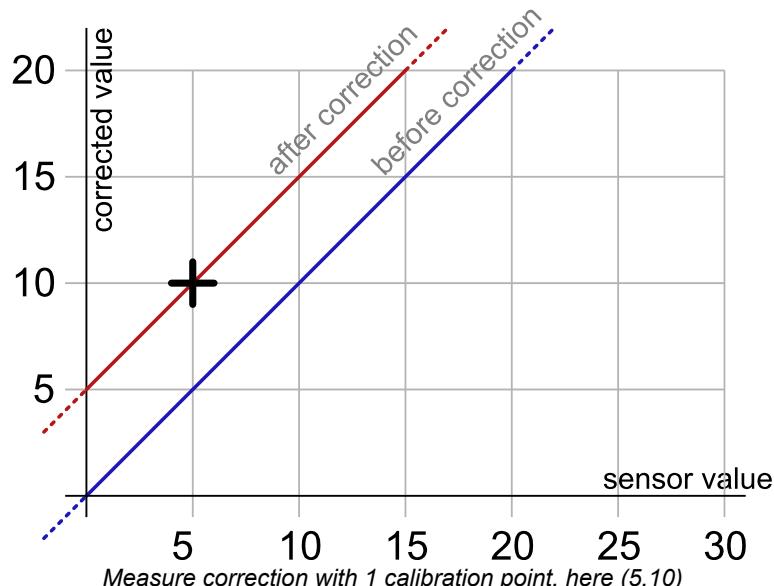
1 to 5 point linear interpolation

These transformations are performed directly inside the Yocto-3D 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 magnetometer function returns b .

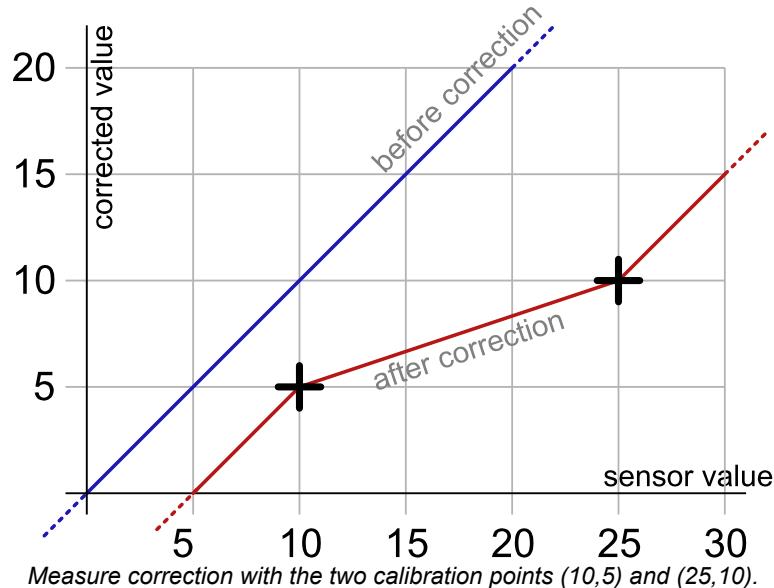


The application is very simple: you only need to call the `calibrateFromPoints()` method of the function you wish to correct. The following code applies the correction illustrated on the graph above to the first magnetometer function found. Note the call to the `saveToFlash` method of the module hosting the function, so that the module does not forget the calibration as soon as it is disconnected.

```
Double[] ValuesBefore = {5};
Double[] ValuesAfter = {10};
YMagnetometer f = YMagnetometer.FirstMagnetometer();
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 magnetometer 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.



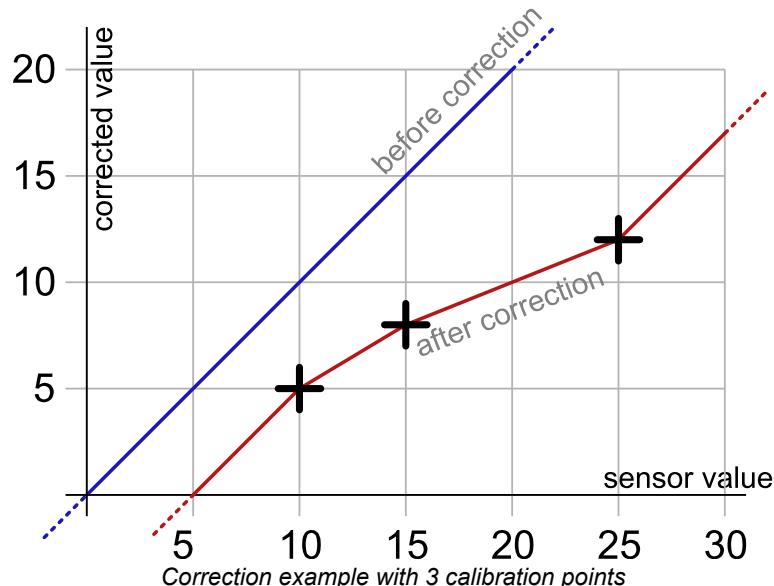
The code allowing you to program this calibration is very similar to the preceding code example.

```
Double[] ValuesBefore = {10,25};
Double[] ValuesAfter  = {5,10};
YMagnetometer f = YMagnetometer.FirstMagnetometer();
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.



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  = {};
YMagnetometer f = YMagnetometer.FirstMagnetometer();
```

```
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.

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.

20. High-level API Reference

This chapter summarizes the high-level API functions to drive your Yocto-3D. 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.

20.1. General functions

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

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

```

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

```

Global functions

`yCheckLogicalName(name)`

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

`yDisableExceptions()`

Disables the use of exceptions to report runtime errors.

`yEnableExceptions()`

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

`yEnableUSBHost(osContext)`

This function is used only on Android.

`yFreeAPI()`

Frees dynamically allocated memory blocks used by the Yoctopuce library.

`yGetAPIVersion()`

Returns the version identifier for the Yoctopuce library in use.

`yGetTickCount()`

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

`yHandleEvents(errmsg)`

Maintains the device-to-library communication channel.

`yInitAPI(mode, errmsg)`

Initializes the Yoctopuce programming library explicitly.

`yPreregisterHub(url, errmsg)`

Fault-tolerant alternative to RegisterHub().

`yRegisterDeviceArrivalCallback(arrivalCallback)`

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

`yRegisterDeviceRemovalCallback(removalCallback)`

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

`yRegisterHub(url, errmsg)`

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

`yRegisterHubDiscoveryCallback(hubDiscoveryCallback)`

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

yRegisterLogFunction(logfun)

Registers a log callback function.

ySelectArchitecture(arch)

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

ySetDelegate(object)

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

ySetTimeout(callback, ms_timeout, arguments)

Invoke the specified callback function after a given timeout.

ySleep(ms_duration, errmsg)

Pauses the execution flow for a specified duration.

yTriggerHubDiscovery(errmsg)

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

yUnregisterHub(url)

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

yUpdateDeviceList(errmsg)

Triggers a (re)detection of connected Yoctopuce modules.

yUpdateDeviceList_async(callback, context)

Triggers a (re)detection of connected Yoctopuce modules.

YAPI.CheckLogicalName() yCheckLogicalName()

YAPI

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

```
js    function yCheckLogicalName( name)
node.js function CheckLogicalName( name)
php   function yCheckLogicalName( $name)
cpp   bool yCheckLogicalName( const string& name)
m     +(BOOL) CheckLogicalName :(NSString *) name
pas   function yCheckLogicalName( name: string): boolean
vb    function yCheckLogicalName( ByVal name As String) As Boolean
cs    bool CheckLogicalName( string name)
java  boolean CheckLogicalName( String name)
py    def 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.DisableExceptions() yDisableExceptions()

YAPI

Disables the use of exceptions to report runtime errors.

```
js function yDisableExceptions( )
node.js function DisableExceptions( )
php function yDisableExceptions( )
cpp void yDisableExceptions( )
m +(void) DisableExceptions
pas procedure yDisableExceptions( )
vb procedure yDisableExceptions( )
cs void DisableExceptions( )
py def 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() yEnableExceptions()

YAPI

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

```
js function yEnableExceptions( )
node.js function EnableExceptions( )
php function yEnableExceptions( )
cpp void yEnableExceptions( )
m +(void) EnableExceptions
pas procedure yEnableExceptions( )
vb procedure yEnableExceptions( )
cs void EnableExceptions( )
py def 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() yEnableUSBHost()

YAPI

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() yFreeAPI()

YAPI

Frees dynamically allocated memory blocks used by the Yoctopuce library.

```
js function yFreeAPI( )
node.js function FreeAPI( )
php function yFreeAPI( )
cpp void yFreeAPI( )
m +(void) FreeAPI
pas procedure yFreeAPI( )
vb procedure yFreeAPI( )
cs void FreeAPI( )
java void FreeAPI( )
py def 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() yGetAPIVersion()

YAPI

Returns the version identifier for the Yoctopuce library in use.

js	function yGetAPIVersion()
nodejs	function GetAPIVersion()
php	function yGetAPIVersion()
cpp	string yGetAPIVersion()
m	+ (NSString*) GetAPIVersion
pas	function yGetAPIVersion(): string
vb	function yGetAPIVersion() As String
cs	String GetAPIVersion()
java	String GetAPIVersion()
py	def 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.GetTickCount() yGetTickCount()

YAPI

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

```
js function yGetTickCount( )
node.js function GetTickCount( )
php function yGetTickCount( )
cpp u64 yGetTickCount( )
m +(u64) GetTickCount
pas function yGetTickCount( ): u64
vb function yGetTickCount( ) As Long
cs ulong GetTickCount( )
java long GetTickCount( )
def 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() yHandleEvents()

YAPI

Maintains the device-to-library communication channel.

```

js   function yHandleEvents( errmsg)
nodejs function HandleEvents( errmsg)
php  function yHandleEvents( &$errmsg)
cpp   YRETCODE yHandleEvents( string& errmsg)
m    +(YRETCODE) HandleEvents :(NSError**) errmsg
pas   function yHandleEvents( var errmsg: string): integer
vb    function yHandleEvents( ByRef errmsg As String) As YRETCODE
cs    YRETCODE HandleEvents( ref string errmsg)
java  int HandleEvents( )
py    def HandleEvents( errmsg=None)

```

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

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

Parameters :

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

YAPI.InitAPI() yInitAPI()

YAPI

Initializes the Yoctopuce programming library explicitly.

js	function yInitAPI (mode, errmsg)
node.js	function InitAPI (mode, errmsg)
php	function yInitAPI (\$mode, &\$errmsg)
cpp	YRETCODE yInitAPI (int mode, string& errmsg)
m	+ (YRETCODE) InitAPI :(int) mode :(NSError**) errmsg
pas	function yInitAPI (mode: integer, var errmsg: string): integer
vb	function yInitAPI (ByVal mode As Integer, ByRef errmsg As String) As Integer
cs	int InitAPI (int mode, ref string errmsg)
java	int InitAPI(int mode)
py	def InitAPI (mode, errmsg=None)

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

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

Parameters :

mode an integer corresponding to the type of automatic device detection to use. Possible values are `Y_DETECT_NONE`, `Y_DETECT_USB`, `Y_DETECT_NET`, and `Y_DETECT_ALL`.

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

YAPI.PreregisterHub() yPreregisterHub()

YAPI

Fault-tolerant alternative to RegisterHub().

```

js   function yPreregisterHub( url, errmsg)
nodejs function PreregisterHub( url, errmsg)
php  function yPreregisterHub( $url, &$errmsg)
cpp   YRETCODE yPreregisterHub( const string& url, string& errmsg)
m    +(YRETCODE) PreregisterHub :NSString * ) url :(NSError**) errmsg
pas   function yPreregisterHub( url: string, var errmsg: string): integer
vb    function yPreregisterHub( ByVal url As String,
                               ByRef errmsg As String) As Integer
cs    int PreregisterHub( string url, ref string errmsg)
java  int PreregisterHub( String url)
py    def PreregisterHub( url, errmsg=None)

```

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

Parameters :

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

Returns :

YAPI_SUCCESS when the call succeeds.

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

YAPI.RegisterDeviceArrivalCallback() yRegisterDeviceArrivalCallback()

YAPI

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

```
js   function yRegisterDeviceArrivalCallback( arrivalCallback)
node.js function RegisterDeviceArrivalCallback( arrivalCallback)
php  function yRegisterDeviceArrivalCallback( $arrivalCallback)
cpp   void yRegisterDeviceArrivalCallback( yDeviceUpdateCallback arrivalCallback)
m    +(void) RegisterDeviceArrivalCallback :(yDeviceUpdateCallback) arrivalCallback
pas   procedure yRegisterDeviceArrivalCallback( arrivalCallback: yDeviceUpdateFunc)
vb    procedure yRegisterDeviceArrivalCallback( ByVal arrivalCallback As yDeviceUpdateFunc)
cs    void RegisterDeviceArrivalCallback( yDeviceUpdateFunc arrivalCallback)
java  void RegisterDeviceArrivalCallback( DeviceArrivalCallback arrivalCallback)
py    def 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() yRegisterDeviceRemovalCallback()

YAPI

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

```

js   function yRegisterDeviceRemovalCallback( removalCallback)
nodejs function RegisterDeviceRemovalCallback( removalCallback)
php  function yRegisterDeviceRemovalCallback( $removalCallback)
cpp   void yRegisterDeviceRemovalCallback( yDeviceUpdateCallback removalCallback)
m    +(void) RegisterDeviceRemovalCallback :(yDeviceUpdateCallback) removalCallback
pas   procedure yRegisterDeviceRemovalCallback( removalCallback: yDeviceUpdateFunc)
vb    procedure yRegisterDeviceRemovalCallback( ByVal removalCallback As yDeviceUpdateFunc)
cs    void RegisterDeviceRemovalCallback( yDeviceUpdateFunc removalCallback)
java  void RegisterDeviceRemovalCallback( DeviceRemovalCallback removalCallback)
py    def 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() yRegisterHub()

YAPI

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

```

js   function yRegisterHub( url, errmsg)
node.js function RegisterHub( url, errmsg)
php  function yRegisterHub( $url, &$errmsg)
cpp   YRETCODE yRegisterHub( const string& url, string& errmsg)
m    +(YRETCODE) RegisterHub :(NSString *) url :(NSError**) errmsg
pas   function yRegisterHub( url: string, var errmsg: string): integer
vb    function yRegisterHub( ByVal url As String,
                           ByRef errmsg As String) As Integer
cs    int RegisterHub( string url, ref string errmsg)
java  int RegisterHub( String url)
py    def RegisterHub( url, errmsg=None)

```

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

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

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

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

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

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

```
http://username:password@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() yRegisterHubDiscoveryCallback()

YAPI

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	procedure yRegisterHubDiscoveryCallback(hubDiscoveryCallback: YHubDiscoveryCallback)
vb	procedure yRegisterHubDiscoveryCallback(ByVal hubDiscoveryCallback As YHubDiscoveryCallback)
cs	void RegisterHubDiscoveryCallback(YHubDiscoveryCallback hubDiscoveryCallback)
java	void RegisterHubDiscoveryCallback(HubDiscoveryCallback hubDiscoveryCallback)
py	def 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, or null

YAPI.RegisterLogFunction() yRegisterLogFunction()

YAPI

Registers a log callback function.

```
cpp void yRegisterLogFunction( yLogFunction logfun)
m +(void) RegisterLogFunction :(yLogCallback) logfun
pas procedure yRegisterLogFunction( logfun: yLogFunc)
vb procedure yRegisterLogFunction( ByVal logfun As yLogFunc)
cs void RegisterLogFunction( yLogFunc logfun)
java void RegisterLogFunction( LogCallback logfun)
py def 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() ySelectArchitecture()

YAPI

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

PY `def SelectArchitecture(arch)`

By default, the Python library automatically detects the appropriate library to use. However, for Linux ARM, it is not possible to reliably distinguish between a Hard Float (armhf) and a Soft Float (armel) install. For 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. Possible values are: "armhf", "armel", "i386", "x86_64", "32bit", "64bit"

Returns :

nothing.

On failure, throws an exception.

YAPI.SetDelegate()
ySetDelegate()**YAPI**

(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.SetTimeout() ySetTimeout()

YAPI

Invoke the specified callback function after a given timeout.

```
js   function ySetTimeout( callback, ms_timeout, arguments )
nodejs function SetTimeout( callback, ms_timeout, arguments )
```

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.
arguments 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.Sleep() ySleep()

YAPI

Pauses the execution flow for a specified duration.

```

js   function ySleep( ms_duration, errmsg)
node.js function Sleep( ms_duration, errmsg)
php  function ySleep( $ms_duration, &$errmsg)
cpp   YRETCODE ySleep( unsigned ms_duration, string& errmsg)
m    +(YRETCODE) Sleep :(unsigned) ms_duration :(NSError **) errmsg
pas   function ySleep( ms_duration: integer, var errmsg: string): integer
vb    function ySleep( ByVal ms_duration As Integer,
                     ByRef errmsg As String) As Integer
cs    int Sleep( int ms_duration, ref string errmsg)
java  int Sleep( long ms_duration)
py    def Sleep( ms_duration, errmsg=None)

```

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

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

Parameters :

`ms_duration` an integer corresponding to the duration of the pause, in milliseconds.

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

YAPI.TriggerHubDiscovery() yTriggerHubDiscovery()

YAPI

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

cpp	YRETCODE yTriggerHubDiscovery(string& errmsg)
m	+ (YRETCODE) TriggerHubDiscovery : (NSError**) errmsg
pas	function yTriggerHubDiscovery(var errmsg: string): integer
vb	function yTriggerHubDiscovery(ByRef errmsg As String) As Integer
cs	int TriggerHubDiscovery(ref string errmsg)
java	int TriggerHubDiscovery()
py	def TriggerHubDiscovery(errmsg=None)

Parameters :

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

Returns :

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

YAPI.UnregisterHub() yUnregisterHub()

YAPI

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

```
js function yUnregisterHub( url)
nodejs function UnregisterHub( url)
php function yUnregisterHub( $url)
cpp void yUnregisterHub( const string& url)
m +(void) UnregisterHub :(NSString *) url
pas procedure yUnregisterHub( url: string)
vb procedure yUnregisterHub( ByVal url As String)
cs void UnregisterHub( string url)
java void UnregisterHub( String url)
py def UnregisterHub( url)
```

Parameters :

url a string containing either "usb" or the

YAPI.UpdateDeviceList() yUpdateDeviceList()

YAPI

Triggers a (re)detection of connected Yoctopuce modules.

js	<code>function yUpdateDeviceList(errmsg)</code>
nodejs	<code>function UpdateDeviceList(errmsg)</code>
php	<code>function yUpdateDeviceList(&\$errmsg)</code>
cpp	<code>YRETCODE yUpdateDeviceList(string& errmsg)</code>
m	<code>+ (YRETCODE) UpdateDeviceList : (NSError**) errmsg</code>
pas	<code>function yUpdateDeviceList(var errmsg: string): integer</code>
vb	<code>function yUpdateDeviceList(ByRef errmsg As String) As YRETCODE</code>
cs	<code>YRETCODE UpdateDeviceList(ref string errmsg)</code>
java	<code>int UpdateDeviceList()</code>
py	<code>def UpdateDeviceList(errmsg=None)</code>

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

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

Parameters :

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

YAPI.UpdateDeviceList_async() yUpdateDeviceList_async()

YAPI

Triggers a (re)detection of connected Yoctopuce modules.

```
js function yUpdateDeviceList_async( callback, context)
nodejs function UpdateDeviceList_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.

20.2. Module control interface

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

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

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

Global functions

yFindModule(func)

Allows you to find a module from its serial number or from its logical name.

yFirstModule()

Starts the enumeration of modules currently accessible.

YModule methods

module→checkFirmware(path, onlynew)

Test if the byn file is valid for this module.

module→describe()

Returns a descriptive text that identifies the module.

module→download(pathname)

Downloads the specified built-in file and returns a binary buffer with its content.

module→functionCount()

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

module→functionId(functionIndex)

Retrieves the hardware identifier of the *n*th function on the module.

module→functionName(functionIndex)

Retrieves the logical name of the *n*th function on the module.

module→functionValue(functionIndex)

Retrieves the advertised value of the *n*th function on the module.

module→get_allSettings()

Returns all the setting of the module.

module→get_beacon()

Returns the state of the localization beacon.

module→get_errorMessage()

Returns the error message of the latest error with this module object.

module→get_errorType()

Returns the numerical error code of the latest error with this module object.

module→get_firmwareRelease()

Returns the version of the firmware embedded in the module.

module→get_hardwareId()

Returns the unique hardware identifier of the module.

module→get_icon2d()

Returns the icon of the module.

module→get_lastLogs()

Returns a string with last logs of the module.

module→get_logicalName()

Returns the logical name of the module.

module→get_luminosity()

Returns the luminosity of the module informative leds (from 0 to 100).

module→get_persistentSettings()

Returns the current state of persistent module settings.

module→get_productId()

Returns the USB device identifier of the module.

module→get_productName()

Returns the commercial name of the module, as set by the factory.

module→get_productRelease()

Returns the hardware release version of the module.

module→get_rebootCountdown()

Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

module→get_serialNumber()

Returns the serial number of the module, as set by the factory.

module→get_upTime()

Returns the number of milliseconds spent since the module was powered on.

module→get_usbCurrent()

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

module→get(userData)

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

module→get(userVar)

Returns the value previously stored in this attribute.

module→isOnline()

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

module→isOnline_async(callback, context)

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

module→load(msValidity)

Preloads the module cache with a specified validity duration.

module→load_async(msValidity, callback, context)

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

module→nextModule()

Continues the module enumeration started using `yFirstModule()`.

module→reboot(secBeforeReboot)

Schedules a simple module reboot after the given number of seconds.

module→registerLogCallback(callback)

Registers a device log callback function.

module→revertFromFlash()

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

module→saveToFlash()

Saves current settings in the nonvolatile memory of the module.

module→set_allSettings(settings)

Restore all the setting of the module.

module→set_beacon(newval)

Turns on or off the module localization beacon.

module→set_logicalName(newval)

Changes the logical name of the module.

module→set_luminosity(newval)

Changes the luminosity of the module informative leds.

module→set_userData(data)

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

module→set_userVar(newval)

Returns the value previously stored in this attribute.

module→triggerFirmwareUpdate(secBeforeReboot)

Schedules a module reboot into special firmware update mode.

module→updateFirmware(path)

Prepare a firmware upgrade of the module.

module→wait_async(callback, context)

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

YModule.FindModule() yFindModule()

YModule

Allows you to find a module from its serial number or from its logical name.

js	function yFindModule(func)
node.js	function FindModule(func)
php	function yFindModule(\$func)
cpp	YModule* yFindModule(string func)
m	+ (YModule*) FindModule : (NSString*) func
pas	function yFindModule(func: string): TYModule
vb	function yFindModule(ByVal func As String) As YModule
cs	YModule FindModule(string func)
java	YModule FindModule(String func)
py	def 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.

Parameters :

func a string containing either the serial number or the logical name of the desired module

Returns :

a `YModule` object allowing you to drive the module or get additional information on the module.

YModule.FirstModule() yFirstModule()

YModule

Starts the enumeration of modules currently accessible.

```
js function yFirstModule( )
nodejs function FirstModule( )
php function yFirstModule( )
cpp YModule* yFirstModule( )
m +(YModule*) FirstModule
pas function yFirstModule( ): TYModule
vb function yFirstModule( ) As YModule
cs YModule FirstModule( )
java YModule FirstModule( )
py def 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→checkFirmware()**YModule**

Test if the byn file is valid for this module.

```

js   function checkFirmware( path, onlynew)
nodejs function checkFirmware( path, onlynew)
cpp   string checkFirmware( string path, bool onlynew)
m    -(NSString*) checkFirmware : (NSString*) path
      : (bool) onlynew

pas  function checkFirmware( path: string, onlynew: boolean): string
vb   function checkFirmware( ) As String
cs   string checkFirmware( string path, bool onlynew)
java  String checkFirmware( String path, boolean onlynew)
py    def checkFirmware( path, onlynew)
cmd   YModule target checkFirmware path onlynew

```

This method is useful to test if the module need to be updated. It's possible to pass an directory instead of a file. In this case this method return the path of the most recent appropriate byn file. If the parameter onlynew is true the function will discard firmware that are older or equal to the installed firmware.

Parameters :

path the path of a byn file or a directory that contain byn files

onlynew return only files that are strictly newer

Returns :

: the path of the byn file to use or a empty string if no byn files match the requirement

On failure, throws an exception or returns a string that start with "error:".

module→describe()

YModule

Returns a descriptive text that identifies the module.

js	function describe ()
node.js	function describe ()
php	function describe ()
cpp	string describe ()
m	-(NSString*) describe
pas	function describe (): string
vb	function describe () As String
cs	string describe ()
java	String describe ()
py	def 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)
nodejs function download( pathname)
php function download( $pathname)
cpp string download( string pathname)
m -(NSMutableData*) download : (NSString*) pathname
pas function download( pathname: string): TByteArray
vb function download( ) As Byte
py def 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→functionCount()

YModule

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

js	function functionCount()
node.js	function functionCount()
php	function functionCount()
cpp	int functionCount()
m	-(int) functionCount
pas	function functionCount() : integer
vb	function functionCount() As Integer
cs	int functionCount()
py	def 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)
nodejs function functionId( functionIndex)
php  function functionId( $functionIndex)
cpp   string functionId( int functionIndex)
m    -(NSString*) functionId : (int) functionIndex
pas   function functionId( functionIndex: integer): string
vb    function functionId( ByVal functionIndex As Integer) As String
cs    string functionId( int functionIndex)
py    def 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)
node.js	function functionName(functionIndex)
php	function functionName(\$functionIndex)
cpp	string functionName(int functionIndex)
m	-(NSString*) functionName : (int) functionIndex
pas	function functionName(functionIndex: integer): string
vb	function functionName(ByVal functionIndex As Integer) As String
cs	string functionName(int functionIndex)
py	def 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→functionValue()**YModule**

Retrieves the advertised value of the *n*th function on the module.

```
js  function functionValue( functionIndex)
nodejs function functionValue( functionIndex)
php  function functionValue( $functionIndex)
cpp   string functionValue( int functionIndex)
m    -(NSString*) functionValue : (int) functionIndex
pas   function functionValue( functionIndex: integer): string
vb    function functionValue( ByVal functionIndex As Integer) As String
cs    string functionValue( int functionIndex)
py    def 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()
module→allSettings()****YModule**

Returns all the setting of the module.

js	function get_allSettings()
node.js	function get_allSettings()
php	function get_allSettings()
cpp	string get_allSettings()
m	-(NSMutableData*) allSettings
pas	function get_allSettings(): TByteArray
vb	function get_allSettings() As Byte
py	def get_allSettings()
cmd	YModule target get_allSettings

Useful to backup all the logical name and calibrations parameters of a connected module.

Returns :

a binary buffer with all settings.

On failure, throws an exception or returns YAPI_INVALID_STRING.

module→get_beacon()**YModule****module→beacon()**

Returns the state of the localization beacon.

js	function get_beacon()
node.js	function get_beacon()
php	function get_beacon()
cpp	Y_BEACON_enum get_beacon()
m	-(Y_BEACON_enum) beacon
pas	function get_beacon() : Integer
vb	function get_beacon() As Integer
cs	int get_beacon()
java	int get_beacon()
py	def 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()
module→errorMessage()**YModule**

Returns the error message of the latest error with this module object.

js	function get_errorMessage()
node.js	function get_errorMessage()
php	function get_errorMessage()
cpp	string get_errorMessage()
m	-(NSString*) errorMessage
pas	function get_errorMessage() : string
vb	function get_errorMessage() As String
cs	string get_errorMessage()
java	String get_errorMessage()
py	def get_errorMessage()

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

Returns :

a string corresponding to the latest error message that occurred while using this 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()
node.js	function get_errorType()
php	function get_errorType()
cpp	YRETCODE get_errorType()
pas	function get_errorType() : YRETCODE
vb	function get_errorType() As YRETCODE
cs	YRETCODE get_errorType()
java	int get_errorType()
py	def get_errorType()

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

Returns :

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

module→get_firmwareRelease()
module→firmwareRelease()**YModule**

Returns the version of the firmware embedded in the module.

js	function get_firmwareRelease()
nodejs	function get_firmwareRelease()
php	function get_firmwareRelease()
cpp	string get_firmwareRelease()
m	-(NSString*) firmwareRelease
pas	function get_firmwareRelease() : string
vb	function get_firmwareRelease() As String
cs	string get_firmwareRelease()
java	String get_firmwareRelease()
py	def 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_hardwareId()
module→hardwareId()****YModule**

Returns the unique hardware identifier of the module.

```
js function get_hardwareId( )
node.js function get_hardwareId( )
php function get_hardwareId( )
cpp string get_hardwareId( )
m -(NSString*) hardwareId
vb function get_hardwareId( ) As String
cs string get_hardwareId( )
java String get_hardwareId( )
py def get_hardwareId( )
```

The unique hardware identifier is made of the device serial number followed by string ".module".

Returns :
a string that uniquely identifies the module

module→get_icon2d()**YModule****module→icon2d()**

Returns the icon of the module.

js	function get_icon2d()
node.js	function get_icon2d()
php	function get_icon2d()
cpp	string get_icon2d()
m	-(NSMutableData*) icon2d
pas	function get_icon2d(): TByteArray
vb	function get_icon2d() As Byte
py	def 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()
module→lastLogs()**YModule**

Returns a string with last logs of the module.

```
js function get_lastLogs( )  
node.js function get_lastLogs( )  
php function get_lastLogs( )  
cpp string get_lastLogs( )  
m -(NSString*) lastLogs  
pas function get_lastLogs( ): string  
vb function get_lastLogs( ) As String  
cs string get_lastLogs( )  
java String get_lastLogs( )  
py def 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()**YModule****module→logicalName()**

Returns the logical name of the module.

```
js   function get_logicalName( )  
nodejs function get_logicalName( )  
php  function get_logicalName( )  
cpp   string get_logicalName( )  
m    -(NSString*) logicalName  
pas   function get_logicalName( ): string  
vb    function get_logicalName( ) As String  
cs    string get_logicalName( )  
java  String get_logicalName( )  
py    def get_logicalName( )  
cmd   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( )
node.js function get_luminosity( )
php function get_luminosity( )
cpp int get_luminosity( )
m -(int) luminosity
pas function get_luminosity( ): LongInt
vb function get_luminosity( ) As Integer
cs int get_luminosity( )
java int get_luminosity( )
py def 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_persistentSettings()
module→persistentSettings()****YModule**

Returns the current state of persistent module settings.

js	function get_persistentSettings()
node.js	function get_persistentSettings()
php	function get_persistentSettings()
cpp	Y_PERSISTENTSETTINGS_enum get_persistentSettings()
m	-(Y_PERSISTENTSETTINGS_enum) persistentSettings
pas	function get_persistentSettings(): Integer
vb	function get_persistentSettings() As Integer
cs	int get_persistentSettings()
java	int get_persistentSettings()
py	def 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) { }
node.js	function get(productId) { }
php	function get(productId) { }
cpp	int get(productId) { }
m	- (int) productId ;
pas	function get(productId) : LongInt;
vb	function get(productId) As Integer;
cs	int get(productId) { }
java	int get(productId) { }
py	def 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()
module→productName()**YModule**

Returns the commercial name of the module, as set by the factory.

js	function get_productName()
nodejs	function get_productName()
php	function get_productName()
cpp	string get_productName()
m	-(NSString*) productName
pas	function get_productName(): string
vb	function get_productName() As String
cs	string get_productName()
java	String get_productName()
py	def 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()
module→productRelease()**YModule**

Returns the hardware release version of the module.

```
js   function get_productRelease( )  
node.js function get_productRelease( )  
php  function get_productRelease( )  
cpp   int get_productRelease( )  
m    -(int) productRelease  
pas  function get_productRelease( ): LongInt  
vb   function get_productRelease( ) As Integer  
cs   int get_productRelease( )  
java  int get_productRelease( )  
py   def get_productRelease( )  
cmd  YModule target get_productRelease
```

Returns :

an integer corresponding to the hardware release version of the module

On failure, throws an exception or returns Y_PRODUCTRELEASE_INVALID.

module→get_rebootCountdown()**YModule****module→rebootCountdown()**

Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

js	function get_rebootCountdown()
nodejs	function get_rebootCountdown()
php	function get_rebootCountdown()
cpp	int get_rebootCountdown()
m	-(int) rebootCountdown
pas	function get_rebootCountdown(): LongInt
vb	function get_rebootCountdown() As Integer
cs	int get_rebootCountdown()
java	int get_rebootCountdown()
py	def 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()
module→serialNumber()**YModule**

Returns the serial number of the module, as set by the factory.

js	function get_serialNumber()
node.js	function get_serialNumber()
php	function get_serialNumber()
cpp	string get_serialNumber()
m	- (NSString*) serialNumber
pas	function get_serialNumber(): string
vb	function get_serialNumber() As String
cs	string get_serialNumber()
java	String get_serialNumber()
py	def 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_upTime()**YModule****module→upTime()**

Returns the number of milliseconds spent since the module was powered on.

js	function get_upTime()
nodejs	function get_upTime()
php	function get_upTime()
cpp	s64 get_upTime()
m	-(s64) upTime
pas	function get_upTime(): int64
vb	function get_upTime() As Long
cs	long get_upTime()
java	long get_upTime()
py	def 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_usbCurrent()
module→usbCurrent()****YModule**

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

js	function get_usbCurrent()
node.js	function get_usbCurrent()
php	function get_usbCurrent()
cpp	int get_usbCurrent()
m	- (int) usbCurrent
pas	function get_usbCurrent() : LongInt
vb	function get_usbCurrent() As Integer
cs	int get_usbCurrent()
java	int get_usbCurrent()
py	def 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	<code>function get(userData) { ... }</code>
nodejs	<code>function get(userData) { ... }</code>
php	<code>function get(userData) { ... }</code>
cpp	<code>void * get(userData) { ... }</code>
m	<code>-(void*)(userData)</code>
pas	<code>function get(userData): Tobject { ... }</code>
vb	<code>function get(userData) As Object { ... }</code>
cs	<code>object get(userData) { ... }</code>
java	<code>Object get(userData) { ... }</code>
py	<code>def get(userData) { ... }</code>

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

Returns :

the object stored previously by the caller.

module→get_userVar() **YModule**
module→userVar()

Returns the value previously stored in this attribute.

```
js function get_userVar( )  
node.js function get_userVar( )  
php function get_userVar( )  
cpp int get_userVar( )  
m -(int) userVar  
pas function get_userVar( ): LongInt  
vb function get_userVar( ) As Integer  
cs int get_userVar( )  
java int get_userVar( )  
py def 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→isOnline()

YModule

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

js	function isOnline ()
node.js	function isOnline ()
php	function isOnline ()
cpp	bool isOnline ()
m	-(BOOL) isOnline
pas	function isOnline (): boolean
vb	function isOnline () As Boolean
cs	bool isOnline ()
java	boolean isOnline ()
py	def isOnline ()

If there 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)
nodejs 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.

<code>js</code>	<code>function load(msValidity)</code>
<code>node.js</code>	<code>function load(msValidity)</code>
<code>php</code>	<code>function load(\$msValidity)</code>
<code>cpp</code>	<code>YRETCODE load(int msValidity)</code>
<code>m</code>	<code>-(YRETCODE) load : (int) msValidity</code>
<code>pas</code>	<code>function load(msValidity: integer): YRETCODE</code>
<code>vb</code>	<code>function load(ByVal msValidity As Integer) As YRETCODE</code>
<code>cs</code>	<code>YRETCODE load(int msValidity)</code>
<code>java</code>	<code>int load(long msValidity)</code>
<code>py</code>	<code>def load(msValidity)</code>

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)
nodejs 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→nextModule()**YModule**

Continues the module enumeration started using `yFirstModule()`.

js	function nextModule()
node.js	function nextModule()
php	function nextModule()
cpp	YModule * nextModule()
m	-(YModule *) nextModule
pas	function nextModule() : TYModule
vb	function nextModule() As YModule
cs	YModule nextModule()
java	YModule nextModule()
py	def nextModule()

Returns :

a pointer to a `YModule` object, corresponding to the next module found, or a `null` pointer if there are no more modules to enumerate.

module→reboot()**YModule**

Schedules a simple module reboot after the given number of seconds.

```
js function reboot( secBeforeReboot)
nodejs function reboot( secBeforeReboot)
php function reboot( $secBeforeReboot)
cpp int reboot( int secBeforeReboot)
m -(int) reboot : (int) secBeforeReboot
pas function reboot( secBeforeReboot: LongInt): LongInt
vb function reboot( ) As Integer
cs int reboot( int secBeforeReboot)
java int reboot( int secBeforeReboot)
py def 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→registerLogCallback()

YModule

Registers a device log callback function.

cpp	void registerLogCallback(YModuleLogCallback callback)
m	- (void) registerLogCallback : (YModuleLogCallback) callback
vb	function registerLogCallback(ByVal callback As YModuleLogCallback) As Integer
cs	int registerLogCallback(LogCallback callback)
java	void registerLogCallback(LogCallback callback)
py	def 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( )
nodejs function revertFromFlash( )
php function revertFromFlash( )
cpp int revertFromFlash( )
m -(int) revertFromFlash
pas function revertFromFlash( ): LongInt
vb function revertFromFlash( ) As Integer
cs int revertFromFlash( )
java int revertFromFlash( )
py def 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()
node.js	function saveToFlash()
php	function saveToFlash()
cpp	int saveToFlash()
m	- (int) saveToFlash
pas	function saveToFlash() : LongInt
vb	function saveToFlash() As Integer
cs	int saveToFlash()
java	int saveToFlash()
py	def 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()
module→setAllSettings()****YModule**

Restore all the setting of the module.

js	function set_allSettings(settings)
node.js	function set_allSettings(settings)
php	function set_allSettings(\$settings)
cpp	int set_allSettings(string settings)
m	-{int) setAllSettings : (NSData*) settings
pas	function set_allSettings(settings: TByteArray): LongInt
vb	procedure set_allSettings()
cs	int set_allSettings()
java	int set_allSettings()
py	def set_allSettings(settings)
cmd	YModule target set_allSettings settings

Useful to restore all the logical name and calibrations parameters of a module from a backup.

Parameters :

settings a binary buffer with all settings.

Returns :

YAPI_SUCCESS when the call succeeds.

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

module→set_beacon()
module→setBeacon()
YModule

Turns on or off the module localization beacon.

js	function set_beacon(newval)
nodejs	function set_beacon(newval)
php	function set_beacon(\$newval)
cpp	int set_beacon(Y_BEACON_enum newval)
m	-(int) setBeacon : (Y_BEACON_enum) newval
pas	function set_beacon(newval: Integer): integer
vb	function set_beacon(ByVal newval As Integer) As Integer
cs	int set_beacon(int newval)
java	int set_beacon(int newval)
py	def 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() module→setLogicalName()

YModule

Changes the logical name of the module.

js	function set_logicalName(newval)
node.js	function set_logicalName(newval)
php	function set_logicalName(\$newval)
cpp	int set_logicalName(const string& newval)
m	- (int) setLogicalName : (NSString*) newval
pas	function set_logicalName(newval: string): integer
vb	function set_logicalName(ByVal newval As String) As Integer
cs	int set_logicalName(string newval)
java	int set_logicalName(String newval)
py	def set_logicalName(newval)
cmd	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)
node.js	function set_luminosity(newval)
php	function set_luminosity(\$newval)
cpp	int set_luminosity(int newval)
m	-(int) setLuminosity : (int) newval
pas	function set_luminosity(newval: LongInt): integer
vb	function set_luminosity(ByVal newval As Integer) As Integer
cs	int set_luminosity(int newval)
java	int set_luminosity(int newval)
py	def 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)
module→setUserData()**YModule**

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

```
js   function set(userData) data
node.js function set(userData) data
php  function set(userData) $data
cpp   void set(userData void* data)
m    -(void) setUserData : (void*) data
pas   procedure set(userData data: Tobject)
vb    procedure set(userData ByVal data As Object)
cs    void set(userData object data)
java  void set(userData Object data)
py    def set(userData data)
```

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

Parameters :

data any kind of object to be stored

module→set_userVar()
module→setUserVar()
YModule

Returns the value previously stored in this attribute.

js	function set_userVar(newval)
node.js	function set_userVar(newval)
php	function set_userVar(\$newval)
cpp	int set_userVar(int newval)
m	-(int) setUserVar : (int) newval
pas	function set_userVar(newval: LongInt): integer
vb	function set_userVar(ByVal newval As Integer) As Integer
cs	int set_userVar(int newval)
java	int set_userVar(int newval)
py	def set_userVar(newval)
cmd	YModule target set_userVar newval

On startup and after a device reboot, the value is always reset to zero.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

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

module→triggerFirmwareUpdate()**YModule**

Schedules a module reboot into special firmware update mode.

```
js function triggerFirmwareUpdate( secBeforeReboot)
nodejs function triggerFirmwareUpdate( secBeforeReboot)
php function triggerFirmwareUpdate( $secBeforeReboot)
cpp int triggerFirmwareUpdate( int secBeforeReboot)
m -(int) triggerFirmwareUpdate : (int) secBeforeReboot
pas function triggerFirmwareUpdate( secBeforeReboot: LongInt): LongInt
vb function triggerFirmwareUpdate( ) As Integer
cs int triggerFirmwareUpdate( int secBeforeReboot)
java int triggerFirmwareUpdate( int secBeforeReboot)
py def 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**

Prepare a firmware upgrade of the module.

<code>js</code>	<code>function updateFirmware(path)</code>
<code>node.js</code>	<code>function updateFirmware(path)</code>
<code>php</code>	<code>function updateFirmware(\$path)</code>
<code>cpp</code>	<code>YFirmwareUpdate updateFirmware(string path)</code>
<code>m</code>	<code>-(YFirmwareUpdate*) updateFirmware : (NSString*) path</code>
<code>pas</code>	<code>function updateFirmware(path: string): TYFirmwareUpdate</code>
<code>vb</code>	<code>function updateFirmware() As YFirmwareUpdate</code>
<code>cs</code>	<code>YFirmwareUpdate updateFirmware(string path)</code>
<code>java</code>	<code>YFirmwareUpdate updateFirmware(String path)</code>
<code>py</code>	<code>def updateFirmware(path)</code>
<code>cmd</code>	<code>YModule target updateFirmware path</code>

This method return a object `YFirmwareUpdate` which will handle the firmware upgrade process.

Parameters :

`path` the path of the byn file to use.

Returns :

: A object `YFirmwareUpdate`.

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)
nodejs function 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.

20.3. Accelerometer function interface

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

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

```

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

```

Global functions

yFindAccelerometer(func)

Retrieves an accelerometer for a given identifier.

yFirstAccelerometer()

Starts the enumeration of accelerometers currently accessible.

YAccelerometer methods

accelerometer→calibrateFromPoints(rawValues, refValues)

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

accelerometer→describe()

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

accelerometer→get_advertisedValue()

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

accelerometer→get_currentRawValue()

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

accelerometer→get_currentValue()

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

accelerometer→get_errorMessage()

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

accelerometer→get_errorType()

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

accelerometer→get_friendlyName()

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

accelerometer→get_functionDescriptor()

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

accelerometer→get_functionId()

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

accelerometer→get_hardwareId()

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

accelerometer→get_highestValue()	Returns the maximal value observed for the acceleration since the device was started.
accelerometer→get_logFrequency()	Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.
accelerometer→get_logicalName()	Returns the logical name of the accelerometer.
accelerometer→get_lowestValue()	Returns the minimal value observed for the acceleration since the device was started.
accelerometer→get_module()	Gets the YModule object for the device on which the function is located.
accelerometer→get_module_async(callback, context)	Gets the YModule object for the device on which the function is located (asynchronous version).
accelerometer→get_recordedData(startTime, endTime)	Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.
accelerometer→get_reportFrequency()	Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.
accelerometer→get_resolution()	Returns the resolution of the measured values.
accelerometer→get_unit()	Returns the measuring unit for the acceleration.
accelerometer→get(userData)	Returns the value of the userData attribute, as previously stored using method set(userData).
accelerometer→get_xValue()	Returns the X component of the acceleration, as a floating point number.
accelerometer→get_yValue()	Returns the Y component of the acceleration, as a floating point number.
accelerometer→get_zValue()	Returns the Z component of the acceleration, as a floating point number.
accelerometer→isOnline()	Checks if the accelerometer is currently reachable, without raising any error.
accelerometer→isOnline_async(callback, context)	Checks if the accelerometer is currently reachable, without raising any error (asynchronous version).
accelerometer→load(msValidity)	Preloads the accelerometer cache with a specified validity duration.
accelerometer→loadCalibrationPoints(rawValues, refValues)	Retrieves error correction data points previously entered using the method calibrateFromPoints.
accelerometer→load_async(msValidity, callback, context)	Preloads the accelerometer cache with a specified validity duration (asynchronous version).
accelerometer→nextAccelerometer()	Continues the enumeration of accelerometers started using yFirstAccelerometer().
accelerometer→registerTimedReportCallback(callback)	Registers the callback function that is invoked on every periodic timed notification.
accelerometer→registerValueCallback(callback)	Registers the callback function that is invoked on every change of advertised value.

accelerometer→set_highestValue(newval)

Changes the recorded maximal value observed.

accelerometer→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

accelerometer→set_logicalName(newval)

Changes the logical name of the accelerometer.

accelerometer→set_lowestValue(newval)

Changes the recorded minimal value observed.

accelerometer→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

accelerometer→set_resolution(newval)

Changes the resolution of the measured physical values.

accelerometer→set_userData(data)

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

accelerometer→wait_async(callback, context)

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

YAccelerometer.FindAccelerometer() yFindAccelerometer()

YAccelerometer

Retrieves an accelerometer for a given identifier.

js	function yFindAccelerometer(func)
node.js	function FindAccelerometer(func)
php	function yFindAccelerometer(\$func)
cpp	YAccelerometer* yFindAccelerometer(const string& func)
m	+ (YAccelerometer*) FindAccelerometer :(NSString*) func
pas	function yFindAccelerometer(func: string): TYAccelerometer
vb	function yFindAccelerometer(ByVal func As String) As YAccelerometer
cs	YAccelerometer FindAccelerometer(string func)
java	YAccelerometer FindAccelerometer(String func)
py	def FindAccelerometer(func)

The identifier can be specified using several formats:

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

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

Parameters :

`func` a string that uniquely characterizes the accelerometer

Returns :

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

YAccelerometer.FirstAccelerometer() yFirstAccelerometer()

YAccelerometer

Starts the enumeration of accelerometers currently accessible.

```
js function yFirstAccelerometer( )
nodejs function FirstAccelerometer( )
php function yFirstAccelerometer( )
cpp YAccelerometer* yFirstAccelerometer( )
m +(YAccelerometer*) FirstAccelerometer
pas function yFirstAccelerometer( ): TYAccelerometer
vb function yFirstAccelerometer( ) As YAccelerometer
cs YAccelerometer FirstAccelerometer( )
java YAccelerometer FirstAccelerometer( )
py def FirstAccelerometer( )
```

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

Returns :

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

accelerometer→calibrateFromPoints()**YAccelerometer**

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)
nodejs function calibrateFromPoints( rawValues, refValues)
php  function calibrateFromPoints( $rawValues, $refValues)
cpp   int calibrateFromPoints( vector<double> rawValues,
                           vector<double> refValues)

m   -(int) calibrateFromPoints : (NSMutableArray*) rawValues
                           : (NSMutableArray*) refValues

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

vb   procedure calibrateFromPoints( )

cs   int calibrateFromPoints( List<double> rawValues,
                           List<double> refValues)

java int calibrateFromPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)

py   def calibrateFromPoints( rawValues, refValues)
cmd  YAccelerometer target calibrateFromPoints rawValues refValues

```

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

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

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

accelerometer→describe()**YAccelerometer**

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

js	function describe ()
node.js	function describe ()
php	function describe ()
cpp	string describe ()
m	-(NSString*) describe
pas	function describe (): string
vb	function describe () As String
cs	string describe ()
java	String describe ()
py	def describe ()

More precisely, **TYPE** is the type of the function, **NAME** is the name used for the first access to the function, **SERIAL** is the serial number of the module if the module is connected or "unresolved", and **FUNCTIONID** is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYL01-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the accelerometer (ex: `Relay(MyCustomName.relay1)=RELAYL01-123456.relay1`)

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

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

```
js function get_advertisedValue( )
node.js function get_advertisedValue( )
php function get_advertisedValue( )
cpp string get_advertisedValue( )
m -(NSString*) advertisedValue
pas function get_advertisedValue( ): string
vb function get_advertisedValue( ) As String
cs string get_advertisedValue( )
java String get_advertisedValue( )
py def get_advertisedValue( )
cmd YAccelerometer target get_advertisedValue
```

Returns :

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

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

accelerometer→get_currentRawValue() accelerometer→currentRawValue()

YAccelerometer

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

js	function get_currentRawValue()
nodejs	function get_currentRawValue()
php	function get_currentRawValue()
cpp	double get_currentRawValue()
m	- (double) currentRawValue
pas	function get_currentRawValue(): double
vb	function get_currentRawValue() As Double
cs	double get_currentRawValue()
java	double get_currentRawValue()
py	def get_currentRawValue()
cmd	YAccelerometer target get_currentRawValue

Returns :

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

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

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

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

```
js function get_currentValue( )
node.js function get_currentValue( )
php function get_currentValue( )
cpp double currentValue( )
m -(double) currentValue
pas function get_currentValue( ): double
vb function get_currentValue( ) As Double
cs double currentValue( )
java double currentValue( )
py def get_currentValue( )
cmd YAccelerometer target get_currentValue
```

Returns :

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

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

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

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

js	function get_errorMessage()
node.js	function get_errorMessage()
php	function get_errorMessage()
cpp	string get_errorMessage()
m	-(NSString*) errorMessage
pas	function get_errorMessage() : string
vb	function get_errorMessage() As String
cs	string get_errorMessage()
java	String get_errorMessage()
py	def get_errorMessage()

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

Returns :

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

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

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

js	function get_errorType()
node.js	function get_errorType()
php	function get_errorType()
cpp	YRETCODE get_errorType()
pas	function get_errorType() : YRETCODE
vb	function get_errorType() As YRETCODE
cs	YRETCODE get_errorType()
java	int get_errorType()
py	def get_errorType()

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

Returns :

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

accelerometer→get_friendlyName() accelerometer→friendlyName()

YAccelerometer

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

js	function get_friendlyName()
nodejs	function get_friendlyName()
php	function get_friendlyName()
cpp	string get_friendlyName()
m	-(NSString*) friendlyName
cs	string get_friendlyName()
java	String get_friendlyName()
py	def get_friendlyName()

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

Returns :

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

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

accelerometer→get_functionDescriptor() accelerometer→functionDescriptor()

YAccelerometer

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

js	function get_functionDescriptor()
node.js	function get_functionDescriptor()
php	function get_functionDescriptor()
cpp	YFUN_DESCR get_functionDescriptor()
m	-(YFUN_DESCR) functionDescriptor
pas	function get_functionDescriptor() : YFUN_DESCR
vb	function get_functionDescriptor() As YFUN_DESCR
cs	YFUN_DESCR get_functionDescriptor()
java	String get_functionDescriptor()
py	def get_functionDescriptor()

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

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

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

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

js	function get_functionId()
node.js	function get_functionId()
php	function get_functionId()
cpp	string get_functionId()
m	-(NSString*) functionId
vb	function get_functionId() As String
cs	string get_functionId()
java	String get_functionId()
py	def get_functionId()

For example `relay1`

Returns :

a string that identifies the accelerometer (ex: `relay1`)

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

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

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

js	function get_hardwareId()
node.js	function get_hardwareId()
php	function get_hardwareId()
cpp	string get_hardwareId()
m	-(NSString*) hardwareId
vb	function get_hardwareId() As String
cs	string get_hardwareId()
java	String get_hardwareId()
py	def get_hardwareId()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the accelerometer (for example RELAYL01-123456.relay1).

Returns :

a string that uniquely identifies the accelerometer (ex: RELAYL01-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

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

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

js	function get_highestValue()
node.js	function get_highestValue()
php	function get_highestValue()
cpp	double get_highestValue()
m	-(double) highestValue
pas	function get_highestValue() : double
vb	function get_highestValue() As Double
cs	double get_highestValue()
java	double get_highestValue()
py	def get_highestValue()
cmd	YAccelerometer target get_highestValue

Returns :

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

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

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

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( )  
nodejs function get_logFrequency( )  
php  function get_logFrequency( )  
cpp   string get_logFrequency( )  
m    -(NSString*) logFrequency  
pas   function get_logFrequency( ):string  
vb    function get_logFrequency( ) As String  
cs    string get_logFrequency( )  
java  String get_logFrequency( )  
py    def get_logFrequency( )  
cmd   YAccelerometer target get_logFrequency
```

Returns :

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

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

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

Returns the logical name of the accelerometer.

js	function get_logicalName()
nodejs	function get_logicalName()
php	function get_logicalName()
cpp	string get_logicalName()
m	-(NSString*) logicalName
pas	function get_logicalName() : string
vb	function get_logicalName() As String
cs	string get_logicalName()
java	String get_logicalName()
py	def get_logicalName()
cmd	YAccelerometer target get_logicalName

Returns :

a string corresponding to the logical name of the accelerometer.

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

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

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

js	function get_lowestValue()
node.js	function get_lowestValue()
php	function get_lowestValue()
cpp	double get_lowestValue()
m	-(double) lowestValue
pas	function get_lowestValue() : double
vb	function get_lowestValue() As Double
cs	double get_lowestValue()
java	double get_lowestValue()
py	def get_lowestValue()
cmd	YAccelerometer target get_lowestValue

Returns :

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

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

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

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

js	function get_module()
nodejs	function get_module()
php	function get_module()
cpp	YModule * get_module()
m	-(YModule*) module
pas	function get_module() : TYModule
vb	function get_module() As YModule
cs	YModule get_module()
java	YModule get_module()
py	def get_module()

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

Returns :

an instance of YModule

accelerometer→get_module_async()
accelerometer→module_async()**YAccelerometer**

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

```
js   function get_module_async( callback, context )
node.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 Javascript 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.

accelerometer→get_recordedData() accelerometer→recordedData()

YAccelerometer

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

js node.js php cpp m	function get_recordedData(startTime, endTime) function get_recordedData(startTime, endTime) function get_recordedData(\$startTime, \$endTime) YDataSet get_recordedData(s64 startTime, s64 endTime) -(YDataSet*) recordedData : (s64) startTime : (s64) endTime
pas vb cs java py cmd	function get_recordedData(startTime: int64, endTime: int64): TYDataSet function get_recordedData() As YDataSet YDataSet get_recordedData(long startTime, long endTime) YDataSet get_recordedData(long startTime, long endTime) def get_recordedData(startTime, endTime) YAccelerometer target get_recordedData startTime endTime

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

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

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any 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.

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

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

js	function get_reportFrequency() ()
nodejs	function get_reportFrequency() ()
php	function get_reportFrequency() ()
cpp	string get_reportFrequency() ()
m	- (NSString*) reportFrequency
pas	function get_reportFrequency() : string
vb	function get_reportFrequency() As String
cs	string get_reportFrequency() ()
java	String get_reportFrequency() ()
py	def get_reportFrequency() ()
cmd	YAccelerometer target get_reportFrequency

Returns :

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

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

accelerometer→get_resolution() accelerometer→resolution()

YAccelerometer

Returns the resolution of the measured values.

js	<code>function get_resolution()</code>
nodejs	<code>function get_resolution()</code>
php	<code>function get_resolution()</code>
cpp	<code>double get_resolution()</code>
m	<code>-(double) resolution</code>
pas	<code>function get_resolution(): double</code>
vb	<code>function get_resolution() As Double</code>
cs	<code>double get_resolution()</code>
java	<code>double get_resolution()</code>
py	<code>def get_resolution()</code>
cmd	<code>YAccelerometer target get_resolution</code>

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

Returns :

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

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

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

Returns the measuring unit for the acceleration.

js	function get_unit()
node.js	function get_unit()
php	function get_unit()
cpp	string get_unit()
m	-(NSString*) unit
pas	function get_unit() : string
vb	function get_unit() As String
cs	string get_unit()
java	String get_unit()
py	def get_unit()
cmd	YAccelerometer target get_unit

Returns :

a string corresponding to the measuring unit for the acceleration

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

accelerometer→get(userData)
accelerometer→userData()**YAccelerometer**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

js	<code>function get(userData) </code>
nodejs	<code>function get(userData) </code>
php	<code>function get(userData) </code>
cpp	<code>void * get(userData) </code>
m	<code>-(void*) userData</code>
pas	<code>function get(userData): Tobject</code>
vb	<code>function get(userData) As Object</code>
cs	<code>object get(userData) </code>
java	<code>Object get(userData) </code>
py	<code>def get(userData) </code>

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

Returns :

the object stored previously by the caller.

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

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

```
js function get_xValue( )
node.js function get_xValue( )
php function get_xValue( )
cpp double get_xValue( )
m -(double) xValue
pas function get_xValue( ): double
vb function get_xValue( ) As Double
cs double get_xValue( )
java double get_xValue( )
py def get_xValue( )
cmd YAccelerometer target get_xValue
```

Returns :

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

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

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

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

js	function get_yValue()
node.js	function get_yValue()
php	function get_yValue()
cpp	double get_yValue()
m	-(double) yValue
pas	function get_yValue() : double
vb	function get_yValue() As Double
cs	double get_yValue()
java	double get_yValue()
py	def get_yValue()
cmd	YAccelerometer target get_yValue

Returns :

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

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

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

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

```
js function get_zValue( )  
node.js function get_zValue( )  
php function get_zValue( )  
cpp double get_zValue( )  
m -(double) zValue  
pas function get_zValue( ): double  
vb function get_zValue( ) As Double  
cs double get_zValue( )  
java double get_zValue( )  
py def get_zValue( )  
cmd YAccelerometer target get_zValue
```

Returns :

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

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

accelerometer→isOnline()**YAccelerometer**

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

js	function isOnline()
node.js	function isOnline()
php	function isOnline()
cpp	bool isOnline()
m	- (BOOL) isOnline
pas	function isOnline() : boolean
vb	function isOnline() As Boolean
cs	bool isOnline()
java	boolean isOnline()
py	def isOnline()

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

Returns :

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

accelerometer→isOnline_async()**YAccelerometer**

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

```
js   function isOnline_async( callback, context )
nodejs function isOnline_async( callback, context )
```

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

accelerometer→load()**YAccelerometer**

Preloads the accelerometer cache with a specified validity duration.

js	<code>function load(msValidity)</code>
node.js	<code>function load(msValidity)</code>
php	<code>function load(\$msValidity)</code>
cpp	<code>YRETCODE load(int msValidity)</code>
m	<code>-(YRETCODE) load : (int) msValidity</code>
pas	<code>function load(msValidity: integer): YRETCODE</code>
vb	<code>function load(ByVal msValidity As Integer) As YRETCODE</code>
cs	<code>YRETCODE load(int msValidity)</code>
java	<code>int load(long msValidity)</code>
py	<code>def load(msValidity)</code>

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

Parameters :

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

accelerometer→loadCalibrationPoints()**YAccelerometer**

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

```

js   function loadCalibrationPoints( rawValues, refValues)
nodejs function loadCalibrationPoints( rawValues, refValues)
php  function loadCalibrationPoints( &$rawValues, &$refValues)
cpp   int loadCalibrationPoints( vector<double>& rawValues,
                                vector<double>& refValues)
m    -(int) loadCalibrationPoints : (NSMutableArray*) rawValues
                           : (NSMutableArray*) refValues
pas  function loadCalibrationPoints( var rawValues: TDoubleArray,
                           var refValues: TDoubleArray): LongInt
vb   procedure loadCalibrationPoints( )
cs   int loadCalibrationPoints( List<double> rawValues,
                           List<double> refValues)
java int loadCalibrationPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)
py   def loadCalibrationPoints( rawValues, refValues)
cmd  YAccelerometer target loadCalibrationPoints rawValues refValues

```

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

accelerometer→load_async()**YAccelerometer**

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

js	function load_async(msValidity, callback, context)
node.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.

accelerometer→nextAccelerometer()**YAccelerometer**

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

js	function nextAccelerometer()
nodejs	function nextAccelerometer()
php	function nextAccelerometer()
cpp	<code>YAccelerometer * nextAccelerometer()</code>
m	<code>-(YAccelerometer*) nextAccelerometer</code>
pas	function nextAccelerometer() : TYAccelerometer
vb	function nextAccelerometer() As YAccelerometer
cs	YAccelerometer nextAccelerometer()
java	<code>YAccelerometer nextAccelerometer()</code>
py	def nextAccelerometer()

Returns :

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

accelerometer→registerTimedReportCallback()**YAccelerometer**

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

js	function registerTimedReportCallback(callback)
node.js	function registerTimedReportCallback(callback)
php	function registerTimedReportCallback(\$callback)
cpp	int registerTimedReportCallback(YAccelerometerTimedReportCallback callback)
m	- (int) registerTimedReportCallback : (YAccelerometerTimedReportCallback) callback
pas	function registerTimedReportCallback(callback : TYAccelerometerTimedReportCallback): LongInt
vb	function registerTimedReportCallback() As Integer
cs	int registerTimedReportCallback(TimedReportCallback callback)
java	int registerTimedReportCallback(TimedReportCallback callback)
py	def 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.

accelerometer→registerValueCallback()**YAccelerometer**

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

<code>js</code>	<code>function registerValueCallback(callback)</code>
<code>nodejs</code>	<code>function registerValueCallback(callback)</code>
<code>php</code>	<code>function registerValueCallback(\$callback)</code>
<code>cpp</code>	<code>int registerValueCallback(YAccelerometerValueCallback callback)</code>
<code>m</code>	<code>- (int) registerValueCallback : (YAccelerometerValueCallback) callback</code>
<code>pas</code>	<code>function registerValueCallback(callback: TYAccelerometerValueCallback): LongInt</code>
<code>vb</code>	<code>function registerValueCallback() As Integer</code>
<code>cs</code>	<code>int registerValueCallback(ValueCallback callback)</code>
<code>java</code>	<code>int registerValueCallback(UpdateCallback callback)</code>
<code>py</code>	<code>def registerValueCallback(callback)</code>

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

Parameters :

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

accelerometer→set_highestValue()
accelerometer→setHighestValue()
YAccelerometer

Changes the recorded maximal value observed.

<code>js</code>	<code>function set_highestValue(newval)</code>
<code>nodejs</code>	<code>function set_highestValue(newval)</code>
<code>php</code>	<code>function set_highestValue(\$newval)</code>
<code>cpp</code>	<code>int set_highestValue(double newval)</code>
<code>m</code>	<code>-(int) setHighestValue : (double) newval</code>
<code>pas</code>	<code>function set_highestValue(newval: double): integer</code>
<code>vb</code>	<code>function set_highestValue(ByVal newval As Double) As Integer</code>
<code>cs</code>	<code>int set_highestValue(double newval)</code>
<code>java</code>	<code>int set_highestValue(double newval)</code>
<code>py</code>	<code>def set_highestValue(newval)</code>
<code>cmd</code>	<code>YAccelerometer target set_highestValue newval</code>

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

Returns :
`YAPI_SUCCESS` if the call succeeds.

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

accelerometer→set_logFrequency() accelerometer→setLogFrequency()

YAccelerometer

Changes the datalogger recording frequency for this function.

js	function set_logFrequency(newval)
node.js	function set_logFrequency(newval)
php	function set_logFrequency(\$newval)
cpp	int set_logFrequency(const string& newval)
m	- (int) setLogFrequency : (NSString*) newval
pas	function set_logFrequency(newval: string): integer
vb	function set_logFrequency(ByVal newval As String) As Integer
cs	int set_logFrequency(string newval)
java	int set_logFrequency(String newval)
py	def set_logFrequency(newval)
cmd	YAccelerometer target set_logFrequency newval

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the logical name of the accelerometer.

js	function set_logicalName(newval)
nodejs	function set_logicalName(newval)
php	function set_logicalName(\$newval)
cpp	int set_logicalName(const string& newval)
m	-(int) setLogicalName : (NSString*) newval
pas	function set_logicalName(newval: string): integer
vb	function set_logicalName(ByVal newval As String) As Integer
cs	int set_logicalName(string newval)
java	int set_logicalName(String newval)
py	def set_logicalName(newval)
cmd	YAccelerometer target set_logicalName newval

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

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

Changes the recorded minimal value observed.

```
js   function set_lowestValue( newval)
node.js function set_lowestValue( newval)
php  function set_lowestValue( $newval)
cpp   int set_lowestValue( double newval)
m    -(int) setLowestValue : (double) newval
pas   function set_lowestValue( newval: double): integer
vb    function set_lowestValue( ByVal newval As Double) As Integer
cs   int set_lowestValue( double newval)
java  int set_lowestValue( double newval)
py    def set_lowestValue( newval)
cmd   YAccelerometer target set_lowestValue newval
```

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

accelerometer→set_reportFrequency() accelerometer→setReportFrequency()

YAccelerometer

Changes the timed value notification frequency for this function.

js	function set_reportFrequency(newval)
nodejs	function set_reportFrequency(newval)
php	function set_reportFrequency(\$newval)
cpp	int set_reportFrequency(const string& newval)
m	-(int) setReportFrequency : (NSString*) newval
pas	function set_reportFrequency(newval: string): integer
vb	function set_reportFrequency(ByVal newval As String) As Integer
cs	int set_reportFrequency(string newval)
java	int set_reportFrequency(String newval)
py	def set_reportFrequency(newval)
cmd	YAccelerometer target set_reportFrequency newval

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

accelerometer→set_resolution() accelerometer→setResolution()

YAccelerometer

Changes the resolution of the measured physical values.

js	function set_resolution(newval)
node.js	function set_resolution(newval)
php	function set_resolution(\$newval)
cpp	int set_resolution(double newval)
m	-{int) setResolution : (double) newval
pas	function set_resolution(newval: double): integer
vb	function set_resolution(ByVal newval As Double) As Integer
cs	int set_resolution(double newval)
java	int set_resolution(double newval)
py	def set_resolution(newval)
cmd	YAccelerometer target set_resolution newval

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

accelerometer→set(userData)

accelerometer→setUserData()

YAccelerometer

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

js	function set(userData)
node.js	function set(userData)
php	function set(userData \$data)
cpp	void set(userData void* data)
m	-(void) setUserData : (void*) data
pas	procedure set(userData: Tobject)
vb	procedure set(userData ByVal data As Object)
cs	void set(userData object data)
java	void set(userData Object data)
py	def set(userData data)

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

Parameters :

data any kind of object to be stored

accelerometer→wait_async()**YAccelerometer**

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

```
js  function wait_async( callback, context)
nodejs function 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.

20.4. Magnetometer function interface

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

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

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

Global functions

yFindMagnetometer(func)

Retrieves a magnetometer for a given identifier.

yFirstMagnetometer()

Starts the enumeration of magnetometers currently accessible.

YMagnetometer methods

magnetometer→calibrateFromPoints(rawValues, refValues)

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

magnetometer→describe()

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

magnetometer→get_advertisedValue()

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

magnetometer→get_currentRawValue()

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

magnetometer→get_currentValue()

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

magnetometer→get_errorMessage()

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

magnetometer→get_errorType()

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

magnetometer→get_friendlyName()

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

magnetometer→get_functionDescriptor()

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

magnetometer→get_functionId()

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

magnetometer→get_hardwareId()

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

magnetometer→get_highestValue()	Returns the maximal value observed for the magnetic field since the device was started.
magnetometer→get_logFrequency()	Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.
magnetometer→get_logicalName()	Returns the logical name of the magnetometer.
magnetometer→get_lowestValue()	Returns the minimal value observed for the magnetic field since the device was started.
magnetometer→get_module()	Gets the YModule object for the device on which the function is located.
magnetometer→get_module_async(callback, context)	Gets the YModule object for the device on which the function is located (asynchronous version).
magnetometer→get_recordedData(startTime, endTime)	Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.
magnetometer→get_reportFrequency()	Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.
magnetometer→get_resolution()	Returns the resolution of the measured values.
magnetometer→get_unit()	Returns the measuring unit for the magnetic field.
magnetometer→get_userData()	Returns the value of the userData attribute, as previously stored using method set(userData).
magnetometer→get_xValue()	Returns the X component of the magnetic field, as a floating point number.
magnetometer→get_yValue()	Returns the Y component of the magnetic field, as a floating point number.
magnetometer→get_zValue()	Returns the Z component of the magnetic field, as a floating point number.
magnetometer→isOnline()	Checks if the magnetometer is currently reachable, without raising any error.
magnetometer→isOnline_async(callback, context)	Checks if the magnetometer is currently reachable, without raising any error (asynchronous version).
magnetometer→load(msValidity)	Preloads the magnetometer cache with a specified validity duration.
magnetometer→loadCalibrationPoints(rawValues, refValues)	Retrieves error correction data points previously entered using the method calibrateFromPoints.
magnetometer→load_async(msValidity, callback, context)	Preloads the magnetometer cache with a specified validity duration (asynchronous version).
magnetometer→nextMagnetometer()	Continues the enumeration of magnetometers started using yFirstMagnetometer().
magnetometer→registerTimedReportCallback(callback)	Registers the callback function that is invoked on every periodic timed notification.
magnetometer→registerValueCallback(callback)	Registers the callback function that is invoked on every change of advertised value.

magnetometer→set_highestValue(newval)

Changes the recorded maximal value observed.

magnetometer→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

magnetometer→set_logicalName(newval)

Changes the logical name of the magnetometer.

magnetometer→set_lowestValue(newval)

Changes the recorded minimal value observed.

magnetometer→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

magnetometer→set_resolution(newval)

Changes the resolution of the measured physical values.

magnetometer→set_userData(data)

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

magnetometer→wait_async(callback, context)

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

Y Magnetometer.FindMagnetometer() yFindMagnetometer()

Y Magnetometer

Retrieves a magnetometer for a given identifier.

js	function yFindMagnetometer(func)
node.js	function FindMagnetometer(func)
php	function yFindMagnetometer(\$func)
cpp	Y Magnetometer* yFindMagnetometer(const string& func)
m	+ (Y Magnetometer*) FindMagnetometer :(NSString*) func
pas	function yFindMagnetometer(func: string): TYMagnetometer
vb	function yFindMagnetometer(ByVal func As String) As YMagnetometer
cs	Y Magnetometer FindMagnetometer(string func)
java	Y Magnetometer FindMagnetometer(String func)
py	def FindMagnetometer(func)

The identifier can be specified using several formats:

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

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

Parameters :

`func` a string that uniquely characterizes the magnetometer

Returns :

a `Y Magnetometer` object allowing you to drive the magnetometer.

Y Magnetometer.FirstMagnetometer() yFirstMagnetometer()

Y Magnetometer

Starts the enumeration of magnetometers currently accessible.

```
js function yFirstMagnetometer( )
nodejs function FirstMagnetometer( )
php function yFirstMagnetometer( )
cpp YMagnetometer* yFirstMagnetometer( )
m +(YMagnetometer*) FirstMagnetometer
pas function yFirstMagnetometer( ): TYMagnetometer
vb function yFirstMagnetometer( ) As YMagnetometer
cs YMagnetometer FirstMagnetometer( )
java YMagnetometer FirstMagnetometer( )
py def FirstMagnetometer( )
```

Use the method `YMagnetometer.nextMagnetometer()` to iterate on next magnetometers.

Returns :

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

magnetometer→calibrateFromPoints()**YMagnetometer**

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)
nodejs function calibrateFromPoints( rawValues, refValues)
php  function calibrateFromPoints( $rawValues, $refValues)
cpp   int calibrateFromPoints( vector<double> rawValues,
                           vector<double> refValues)

m   -(int) calibrateFromPoints : (NSMutableArray*) rawValues
                           : (NSMutableArray*) refValues

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

vb   procedure calibrateFromPoints( )

cs   int calibrateFromPoints( List<double> rawValues,
                           List<double> refValues)

java int calibrateFromPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)

py   def calibrateFromPoints( rawValues, refValues)
cmd  YMagnetometer target calibrateFromPoints rawValues refValues

```

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

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

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

magnetometer→describe()**YMagnetometer**

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

js	function describe ()
node.js	function describe ()
php	function describe ()
cpp	string describe ()
m	-(NSString*) describe
pas	function describe (): string
vb	function describe () As String
cs	string describe ()
java	String describe ()
py	def describe ()

More precisely, **TYPE** is the type of the function, **NAME** is the name used for the first access to the function, **SERIAL** is the serial number of the module if the module is connected or "unresolved", and **FUNCTIONID** is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYL01-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the magnetometer (ex: `Relay(MyCustomName.relay1)=RELAYL01-123456.relay1`)

magnetometer→get_advertisedValue()
magnetometer→advertisedValue()**YMagnetometer**

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

```
js function get_advertisedValue( )
node.js function get_advertisedValue( )
php function get_advertisedValue( )
cpp string get_advertisedValue( )
m -(NSString*) advertisedValue
pas function get_advertisedValue( ): string
vb function get_advertisedValue( ) As String
cs string get_advertisedValue( )
java String get_advertisedValue( )
py def get_advertisedValue( )
cmd YMagnetometer target get_advertisedValue
```

Returns :

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

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

magnetometer→get_currentRawValue() magnetometer→currentRawValue()

YMagnetometer

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

js	function get_currentRawValue()
nodejs	function get_currentRawValue()
php	function get_currentRawValue()
cpp	double get_currentRawValue()
m	-(double) currentRawValue
pas	function get_currentRawValue() : double
vb	function get_currentRawValue() As Double
cs	double get_currentRawValue()
java	double get_currentRawValue()
py	def get_currentRawValue()
cmd	Y Magnetometer target get_currentRawValue

Returns :

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

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

magnetometer→get_currentValue()
magnetometer→currentValue()**YMagnetometer**

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

```
js function get_currentValue( )
node.js function get_currentValue( )
php function get_currentValue( )
cpp double get_currentValue( )
m -(double) currentValue
pas function get_currentValue( ): double
vb function get_currentValue( ) As Double
cs double get_currentValue( )
java double get_currentValue( )
py def get_currentValue( )
cmd YMagnetometer target get_currentValue
```

Returns :

a floating point number corresponding to the current value of the magnetic field, in mT, as a floating point number

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

magnetometer→getErrorMessage()
magnetometer→errorMessage()**YMagnetometer**

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

js	function getErrorMessage()
node.js	function getErrorMessage()
php	function getErrorMessage()
cpp	string getErrorMessage()
m	-(NSString*) errorMessage
pas	function getErrorMessage() : string
vb	function getErrorMessage() As String
cs	string getErrorMessage()
java	String getErrorMessage()
py	def getErrorMessage()

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

Returns :

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

magnetometer→get_errorType()
magnetometer→errorType()**YMagnetometer**

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

js	function get_errorType()
node.js	function get_errorType()
php	function get_errorType()
cpp	YRETCODE get_errorType()
pas	function get_errorType() : YRETCODE
vb	function get_errorType() As YRETCODE
cs	YRETCODE get_errorType()
java	int get_errorType()
py	def get_errorType()

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

Returns :

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

magnetometer→get_friendlyName() magnetometer→friendlyName()

YMagnetometer

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

js	function get_friendlyName()
nodejs	function get_friendlyName()
php	function get_friendlyName()
cpp	string get_friendlyName()
m	-(NSString*) friendlyName
cs	string get_friendlyName()
java	String get_friendlyName()
py	def get_friendlyName()

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

Returns :

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

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

magnetometer→get_functionDescriptor()**YMagnetometer****magnetometer→functionDescriptor()**

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

js	function get_functionDescriptor()
node.js	function get_functionDescriptor()
php	function get_functionDescriptor()
cpp	YFUN_DESCR get_functionDescriptor()
m	-(YFUN_DESCR) functionDescriptor
pas	function get_functionDescriptor() : YFUN_DESCR
vb	function get_functionDescriptor() As YFUN_DESCR
cs	YFUN_DESCR get_functionDescriptor()
java	String get_functionDescriptor()
py	def get_functionDescriptor()

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

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

magnetometer→get_functionId()
magnetometer→functionId()**YMagnetometer**

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

js	function get_functionId()
node.js	function get_functionId()
php	function get_functionId()
cpp	string get_functionId()
m	-(NSString*) functionId
vb	function get_functionId() As String
cs	string get_functionId()
java	String get_functionId()
py	def get_functionId()

For example `relay1`

Returns :

a string that identifies the magnetometer (ex: `relay1`)

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

magnetometer→get_hardwareId()
magnetometer→hardwareId()**YMagnetometer**

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

js	function get_hardwareId()
node.js	function get_hardwareId()
php	function get_hardwareId()
cpp	string get_hardwareId()
m	-(NSString*) hardwareId
vb	function get_hardwareId() As String
cs	string get_hardwareId()
java	String get_hardwareId()
py	def get_hardwareId()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the magnetometer (for example RELAYL01-123456.relay1).

Returns :

a string that uniquely identifies the magnetometer (ex: RELAYL01-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

magnetometer→get_highestValue()
magnetometer→highestValue()**YMagnetometer**

Returns the maximal value observed for the magnetic field since the device was started.

js	function get_highestValue()
node.js	function get_highestValue()
php	function get_highestValue()
cpp	double get_highestValue()
m	-(double) highestValue
pas	function get_highestValue() : double
vb	function get_highestValue() As Double
cs	double get_highestValue()
java	double get_highestValue()
py	def get_highestValue()
cmd	YMagnetometer target get_highestValue

Returns :

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

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

magnetometer→get_logFrequency() magnetometer→logFrequency()

YMagnetometer

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()
nodejs	function get_logFrequency()
php	function get_logFrequency()
cpp	string get_logFrequency()
m	-(NSString*) logFrequency
pas	function get_logFrequency() : string
vb	function get_logFrequency() As String
cs	string get_logFrequency()
java	String get_logFrequency()
py	def get_logFrequency()
cmd	Y Magnetometer target get_logFrequency

Returns :

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

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

magnetometer→get_logicalName()
magnetometer→logicalName()**YMagnetometer**

Returns the logical name of the magnetometer.

js	function get_logicalName()
node.js	function get_logicalName()
php	function get_logicalName()
cpp	string get_logicalName()
m	-(NSString*) logicalName
pas	function get_logicalName() : string
vb	function get_logicalName() As String
cs	string get_logicalName()
java	String get_logicalName()
py	def get_logicalName()
cmd	YMagnetometer target get_logicalName

Returns :

a string corresponding to the logical name of the magnetometer.

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

magnetometer→get_lowestValue()
magnetometer→lowestValue()**YMagnetometer**

Returns the minimal value observed for the magnetic field since the device was started.

js	function get_lowestValue()
node.js	function get_lowestValue()
php	function get_lowestValue()
cpp	double get_lowestValue()
m	-(double) lowestValue
pas	function get_lowestValue() : double
vb	function get_lowestValue() As Double
cs	double get_lowestValue()
java	double get_lowestValue()
py	def get_lowestValue()
cmd	Y Magnetometer target get_lowestValue

Returns :

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

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

magnetometer→get_module()
magnetometer→module()**YMagnetometer**

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

js	function get_module()
nodejs	function get_module()
php	function get_module()
cpp	YModule * get_module()
m	-(YModule*) module
pas	function get_module() : TYModule
vb	function get_module() As YModule
cs	YModule get_module()
java	YModule get_module()
py	def get_module()

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

Returns :

an instance of YModule

magnetometer→get_module_async()
magnetometer→module_async()**YMagnetometer**

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

`js` `function get_module_async(callback, context)`
`node.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 Javascript 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.

magnetometer→get_recordedData() magnetometer→recordedData()

YMagnetometer

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

js node.js php cpp m	function get_recordedData(startTime, endTime) function get_recordedData(startTime, endTime) function get_recordedData(\$startTime, \$endTime) YDataSet get_recordedData(s64 startTime, s64 endTime) -(YDataSet*) recordedData : (s64) startTime : (s64) endTime
pas vb cs java py cmd	function get_recordedData(startTime: int64, endTime: int64): TYDataSet function get_recordedData() As YDataSet YDataSet get_recordedData(long startTime, long endTime) YDataSet get_recordedData(long startTime, long endTime) def get_recordedData(startTime, endTime) YMagnetometer target get_recordedData startTime endTime

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

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

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any 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.

magnetometer→get_reportFrequency()
magnetometer→reportFrequency()**YMagnetometer**

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

```
js   function get_reportFrequency( )  
nodejs function get_reportFrequency( )  
php  function get_reportFrequency( )  
cpp   string get_reportFrequency( )  
m    -(NSString*) reportFrequency  
pas   function get_reportFrequency( ): string  
vb    function get_reportFrequency( ) As String  
cs    string get_reportFrequency( )  
java  String get_reportFrequency( )  
py    def get_reportFrequency( )  
cmd   YMagnetometer target get_reportFrequency
```

Returns :

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

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

magnetometer→get_resolution()
magnetometer→resolution()**YMagnetometer**

Returns the resolution of the measured values.

js	function get_resolution()
node.js	function get_resolution()
php	function get_resolution()
cpp	double get_resolution()
m	-(double) resolution
pas	function get_resolution(): double
vb	function get_resolution() As Double
cs	double get_resolution()
java	double get_resolution()
py	def get_resolution()
cmd	YMagnetometer target get_resolution

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

Returns :

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

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

magnetometer→get_unit()
magnetometer→unit()**YMagnetometer**

Returns the measuring unit for the magnetic field.

js	function get_unit()
node.js	function get_unit()
php	function get_unit()
cpp	string get_unit()
m	-(NSString*) unit
pas	function get_unit() : string
vb	function get_unit() As String
cs	string get_unit()
java	String get_unit()
py	def get_unit()
cmd	YMagnetometer target get_unit

Returns :

a string corresponding to the measuring unit for the magnetic field

On failure, throws an exception or returns Y_UNIT_INVALID.

magnetometer→get(userData)
magnetometer→userData()**YMagnetometer**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

js	<code>function get(userData) </code>
nodejs	<code>function get(userData) </code>
php	<code>function get(userData) </code>
cpp	<code>void * get(userData) </code>
m	<code>-(void*) userData</code>
pas	<code>function get(userData): Tobject</code>
vb	<code>function get(userData) As Object</code>
cs	<code>object get(userData) </code>
java	<code>Object get(userData) </code>
py	<code>def get(userData) </code>

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

Returns :

the object stored previously by the caller.

magnetometer→get_xValue()
magnetometer→xValue()**YMagnetometer**

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

```
js function get_xValue( )
node.js function get_xValue( )
php function get_xValue( )
cpp double get_xValue( )
m -(double) xValue
pas function get_xValue( ): double
vb function get_xValue( ) As Double
cs double get_xValue( )
java double get_xValue( )
py def get_xValue( )
cmd YMagnetometer target get_xValue
```

Returns :

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

On failure, throws an exception or returns Y_XVALUE_INVALID.

magnetometer→get_yValue()**magnetometer→yValue()****YMagnetometer**

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

js	function get_yValue()
node.js	function get_yValue()
php	function get_yValue()
cpp	double get_yValue()
m	-(double) yValue
pas	function get_yValue() : double
vb	function get_yValue() As Double
cs	double get_yValue()
java	double get_yValue()
py	def get_yValue()
cmd	YMagnetometer target get_yValue

Returns :

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

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

magnetometer→get_zValue()
magnetometer→zValue()**YMagnetometer**

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

```
js function get_zValue( )  
node.js function get_zValue( )  
php function get_zValue( )  
cpp double get_zValue( )  
m -(double) zValue  
pas function get_zValue( ): double  
vb function get_zValue( ) As Double  
cs double get_zValue( )  
java double get_zValue( )  
py def get_zValue( )  
cmd YMagnetometer target get_zValue
```

Returns :

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

On failure, throws an exception or returns Y_ZVALUE_INVALID.

magnetometer→isOnline()**YMagnetometer**

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

js	function isOnline()
node.js	function isOnline()
php	function isOnline()
cpp	bool isOnline()
m	- (BOOL) isOnline
pas	function isOnline() : boolean
vb	function isOnline() As Boolean
cs	bool isOnline()
java	boolean isOnline()
py	def isOnline()

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

Returns :

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

magnetometer→isOnline_async()**YMagnetometer**

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

```
js   function isOnline_async( callback, context )
nodejs function isOnline_async( callback, context )
```

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

magnetometer→load()**YMagnetometer**

Preloads the magnetometer cache with a specified validity duration.

js	<code>function load(msValidity)</code>
node.js	<code>function load(msValidity)</code>
php	<code>function load(\$msValidity)</code>
cpp	<code>YRETCODE load(int msValidity)</code>
m	<code>-(YRETCODE) load : (int) msValidity</code>
pas	<code>function load(msValidity: integer): YRETCODE</code>
vb	<code>function load(ByVal msValidity As Integer) As YRETCODE</code>
cs	<code>YRETCODE load(int msValidity)</code>
java	<code>int load(long msValidity)</code>
py	<code>def load(msValidity)</code>

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

Parameters :

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

magnetometer→loadCalibrationPoints()**YMagnetometer**

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

```

js   function loadCalibrationPoints( rawValues, refValues)
nodejs function loadCalibrationPoints( rawValues, refValues)
php  function loadCalibrationPoints( &$rawValues, &$refValues)
cpp   int loadCalibrationPoints( vector<double>& rawValues,
                                vector<double>& refValues)
m    -(int) loadCalibrationPoints : (NSMutableArray*) rawValues
                           : (NSMutableArray*) refValues
pas   function loadCalibrationPoints( var rawValues: TDoubleArray,
                           var refValues: TDoubleArray): LongInt
vb    procedure loadCalibrationPoints( )
cs    int loadCalibrationPoints( List<double> rawValues,
                           List<double> refValues)
java  int loadCalibrationPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)
py    def loadCalibrationPoints( rawValues, refValues)
cmd   YMagnetometer target loadCalibrationPoints rawValues refValues

```

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

magnetometer→load_async()**YMagnetometer**

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

js	function load_async(msValidity, callback, context)
node.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.

magnetometer→nextMagnetometer()**Y Magnetometer**

Continues the enumeration of magnetometers started using `yFirstMagnetometer()`.

js	function nextMagnetometer()
nodejs	function nextMagnetometer()
php	function nextMagnetometer()
cpp	<code>Y Magnetometer * nextMagnetometer()</code>
m	<code>-(Y Magnetometer*) nextMagnetometer</code>
pas	function nextMagnetometer() : TY Magnetometer
vb	function nextMagnetometer() As Y Magnetometer
cs	Y Magnetometer nextMagnetometer()
java	Y Magnetometer nextMagnetometer()
py	def nextMagnetometer()

Returns :

a pointer to a `Y Magnetometer` object, corresponding to a magnetometer currently online, or a null pointer if there are no more magnetometers to enumerate.

magnetometer→registerTimedReportCallback()**YMagnetometer**

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

js	<code>function registerTimedReportCallback(callback)</code>
node.js	<code>function registerTimedReportCallback(callback)</code>
php	<code>function registerTimedReportCallback(\$callback)</code>
cpp	<code>int registerTimedReportCallback(YMagnetometerTimedReportCallback callback)</code>
m	<code>-(int) registerTimedReportCallback : (YMagnetometerTimedReportCallback) callback</code>
pas	<code>function registerTimedReportCallback(callback: TYMagnetometerTimedReportCallback): LongInt</code>
vb	<code>function registerTimedReportCallback() As Integer</code>
cs	<code>int registerTimedReportCallback(TimedReportCallback callback)</code>
java	<code>int registerTimedReportCallback(TimedReportCallback callback)</code>
py	<code>def registerTimedReportCallback(callback)</code>

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

Parameters :

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

magnetometer→registerValueCallback()**YMagnetometer**

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

js	function registerValueCallback(callback)
nodejs	function registerValueCallback(callback)
php	function registerValueCallback(\$callback)
cpp	int registerValueCallback(YMagnetometerValueCallback callback)
m	-(int) registerValueCallback : (YMagnetometerValueCallback) callback
pas	function registerValueCallback(callback: TYMagnetometerValueCallback): LongInt
vb	function registerValueCallback() As Integer
cs	int registerValueCallback(ValueCallback callback)
java	int registerValueCallback(UpdateCallback callback)
py	def 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.

magnetometer→set_highestValue()
magnetometer→setHighestValue()**YMagnetometer**

Changes the recorded maximal value observed.

<code>js</code>	<code>function set_highestValue(newval)</code>
<code>node.js</code>	<code>function set_highestValue(newval)</code>
<code>php</code>	<code>function set_highestValue(\$newval)</code>
<code>cpp</code>	<code>int set_highestValue(double newval)</code>
<code>m</code>	<code>-(int) setHighestValue : (double) newval</code>
<code>pas</code>	<code>function set_highestValue(newval: double): integer</code>
<code>vb</code>	<code>function set_highestValue(ByVal newval As Double) As Integer</code>
<code>cs</code>	<code>int set_highestValue(double newval)</code>
<code>java</code>	<code>int set_highestValue(double newval)</code>
<code>py</code>	<code>def set_highestValue(newval)</code>
<code>cmd</code>	<code>YMagnetometer target set_highestValue newval</code>

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

magnetometer→set_logFrequency()
magnetometer→setLogFrequency()**YMagnetometer**

Changes the datalogger recording frequency for this function.

js	function set_logFrequency(newval)
node.js	function set_logFrequency(newval)
php	function set_logFrequency(\$newval)
cpp	int set_logFrequency(const string& newval)
m	- (int) setLogFrequency : (NSString*) newval
pas	function set_logFrequency(newval: string): integer
vb	function set_logFrequency(ByVal newval As String) As Integer
cs	int set_logFrequency(string newval)
java	int set_logFrequency(String newval)
py	def set_logFrequency(newval)
cmd	Y Magnetometer target set_logFrequency newval

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

magnetometer→set_logicalName()
magnetometer→setLogicalName()**YMagnetometer**

Changes the logical name of the magnetometer.

js	function set_logicalName(newval)
nodejs	function set_logicalName(newval)
php	function set_logicalName(\$newval)
cpp	int set_logicalName(const string& newval)
m	-(int) setLogicalName : (NSString*) newval
pas	function set_logicalName(newval: string): integer
vb	function set_logicalName(ByVal newval As String) As Integer
cs	int set_logicalName(string newval)
java	int set_logicalName(String newval)
py	def set_logicalName(newval)
cmd	YMagnetometer target set_logicalName newval

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

magnetometer→set_lowestValue() magnetometer→setLowestValue()

YMagnetometer

Changes the recorded minimal value observed.

js	function set_lowestValue(newval)
node.js	function set_lowestValue(newval)
php	function set_lowestValue(\$newval)
cpp	int set_lowestValue(double newval)
m	-{int) setLowestValue : (double) newval
pas	function set_lowestValue(newval: double): integer
vb	function set_lowestValue(ByVal newval As Double) As Integer
cs	int set_lowestValue(double newval)
java	int set_lowestValue(double newval)
py	def set_lowestValue(newval)
cmd	Y Magnetometer target set_lowestValue newval

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

magnetometer→set_reportFrequency() magnetometer→setReportFrequency()

YMagnetometer

Changes the timed value notification frequency for this function.

js	function set_reportFrequency(newval)
nodejs	function set_reportFrequency(newval)
php	function set_reportFrequency(\$newval)
cpp	int set_reportFrequency(const string& newval)
m	-(int) setReportFrequency : (NSString*) newval
pas	function set_reportFrequency(newval: string): integer
vb	function set_reportFrequency(ByVal newval As String) As Integer
cs	int set_reportFrequency(string newval)
java	int set_reportFrequency(String newval)
py	def set_reportFrequency(newval)
cmd	YMagnetometer target set_reportFrequency newval

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

magnetometer→set_resolution() magnetometer→setResolution()

YMagnetometer

Changes the resolution of the measured physical values.

js	<code>function set_resolution(newval)</code>
node.js	<code>function set_resolution(newval)</code>
php	<code>function set_resolution(\$newval)</code>
cpp	<code>int set_resolution(double newval)</code>
m	<code>-(int) setResolution : (double) newval</code>
pas	<code>function set_resolution(newval: double): integer</code>
vb	<code>function set_resolution(ByVal newval As Double) As Integer</code>
cs	<code>int set_resolution(double newval)</code>
java	<code>int set_resolution(double newval)</code>
py	<code>def set_resolution(newval)</code>
cmd	<code>Y Magnetometer target set_resolution newval</code>

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

magnetometer→set(userData)
magnetometer→setUserData()
YMagnetometer

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

js	function set(userData)
node.js	function set(userData)
php	function set(userData \$data)
cpp	void set(userData void* data)
m	-(void) setUserData : (void*) data
pas	procedure set(userData: TObject)
vb	procedure set(userData ByVal data As Object)
cs	void set(userData object data)
java	void set(userData Object data)
py	def set(userData data)

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

Parameters :

data any kind of object to be stored

magnetometer→wait_async()**YMagnetometer**

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

```
js   function wait_async( callback, context)
nodejs function 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.

20.5. Reference frame configuration

This class is used to setup the base orientation of the Yocto-3D, so that the orientation functions, relative to the earth surface plane, use the proper reference frame. The class also implements a tridimensional sensor calibration process, which can compensate for local variations of standard gravity and improve the precision of the tilt sensors.

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

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

Global functions

yFindRefFrame(func)

Retrieves a reference frame for a given identifier.

yFirstRefFrame()

Starts the enumeration of reference frames currently accessible.

YRefFrame methods

refframe→cancel3DCalibration()

Aborts the sensors tridimensional calibration process et restores normal settings.

refframe→describe()

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

refframe→get_3DCalibrationHint()

Returns instructions to proceed to the tridimensional calibration initiated with method start3DCalibration.

refframe→get_3DCalibrationLogMsg()

Returns the latest log message from the calibration process.

refframe→get_3DCalibrationProgress()

Returns the global process indicator for the tridimensional calibration initiated with method start3DCalibration.

refframe→get_3DCalibrationStage()

Returns index of the current stage of the calibration initiated with method start3DCalibration.

refframe→get_3DCalibrationStageProgress()

Returns the process indicator for the current stage of the calibration initiated with method start3DCalibration.

refframe→get_advertisedValue()

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

refframe→get_bearing()

Returns the reference bearing used by the compass.

reframe→get_errorMessage()	Returns the error message of the latest error with the reference frame.
reframe→get_errorType()	Returns the numerical error code of the latest error with the reference frame.
reframe→get_friendlyName()	Returns a global identifier of the reference frame in the format MODULE_NAME . FUNCTION_NAME.
reframe→get_functionDescriptor()	Returns a unique identifier of type YFUN_DESCR corresponding to the function.
reframe→get_functionId()	Returns the hardware identifier of the reference frame, without reference to the module.
reframe→get_hardwareId()	Returns the unique hardware identifier of the reference frame in the form SERIAL . FUNCTIONID.
reframe→get_logicalName()	Returns the logical name of the reference frame.
reframe→get_module()	Gets the YModule object for the device on which the function is located.
reframe→get_module_async(callback, context)	Gets the YModule object for the device on which the function is located (asynchronous version).
reframe→get_mountOrientation()	Returns the installation orientation of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.
reframe→get_mountPosition()	Returns the installation position of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.
reframe→get(userData)	Returns the value of the userData attribute, as previously stored using method set(userData).
reframe→isOnline()	Checks if the reference frame is currently reachable, without raising any error.
reframe→isOnline_async(callback, context)	Checks if the reference frame is currently reachable, without raising any error (asynchronous version).
reframe→load(msValidity)	Preloads the reference frame cache with a specified validity duration.
reframe→load_async(msValidity, callback, context)	Preloads the reference frame cache with a specified validity duration (asynchronous version).
reframe→more3DCalibration()	Continues the sensors tridimensional calibration process previously initiated using method start3DCalibration().
reframe→nextRefFrame()	Continues the enumeration of reference frames started using yFirstRefFrame().
reframe→registerValueCallback(callback)	Registers the callback function that is invoked on every change of advertised value.
reframe→save3DCalibration()	Applies the sensors tridimensional calibration parameters that have just been computed.
reframe→set_bearing(newval)	Changes the reference bearing used by the compass.
reframe→set_logicalName(newval)	

Changes the logical name of the reference frame.

refframe→set_mountPosition(position, orientation)

Changes the compass and tilt sensor frame of reference.

refframe→set(userData)

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

refframe→start3DCalibration()

Initiates the sensors tridimensional calibration process.

refframe→wait_async(callback, context)

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

YRefFrame.FindRefFrame() yFindRefFrame()

YRefFrame

Retrieves a reference frame for a given identifier.

js	function yFindRefFrame(func)
node.js	function FindRefFrame(func)
php	function yFindRefFrame(\$func)
cpp	YRefFrame* yFindRefFrame(const string& func)
m	+(YRefFrame*) FindRefFrame :(NSString*) func
pas	function yFindRefFrame(func: string): TYRefFrame
vb	function yFindRefFrame(ByVal func As String) As YRefFrame
cs	YRefFrame FindRefFrame(string func)
java	YRefFrame FindRefFrame(String func)
py	def FindRefFrame(func)

The identifier can be specified using several formats:

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

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

Parameters :

`func` a string that uniquely characterizes the reference frame

Returns :

a `YRefFrame` object allowing you to drive the reference frame.

YRefFrame.FirstRefFrame() yFirstRefFrame()

YRefFrame

Starts the enumeration of reference frames currently accessible.

js	function yFirstRefFrame()
node.js	function FirstRefFrame()
php	function yFirstRefFrame()
cpp	YRefFrame* yFirstRefFrame()
m	+(YRefFrame*) FirstRefFrame
pas	function yFirstRefFrame() : TYRefFrame
vb	function yFirstRefFrame() As YRefFrame
cs	YRefFrame FirstRefFrame()
java	YRefFrame FirstRefFrame()
py	def FirstRefFrame()

Use the method `YRefFrame.nextRefFrame()` to iterate on next reference frames.

Returns :

a pointer to a `YRefFrame` object, corresponding to the first reference frame currently online, or a `null` pointer if there are none.

reframe→cancel3DCalibration()**YRefFrame**

Aborts the sensors tridimensional calibration process et restores normal settings.

```
js function cancel3DCalibration( )  
nodejs function cancel3DCalibration( )  
php function cancel3DCalibration( )  
cpp int cancel3DCalibration( )  
m -(int) cancel3DCalibration  
pas function cancel3DCalibration( ): LongInt  
vb function cancel3DCalibration( ) As Integer  
cs int cancel3DCalibration( )  
java int cancel3DCalibration( )  
py def cancel3DCalibration( )  
cmd YRefFrame target cancel3DCalibration
```

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

refframe→describe()**YRefFrame**

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

js	function describe ()
nodejs	function describe ()
php	function describe ()
cpp	string describe ()
m	-(NSString*) describe
pas	function describe (): string
vb	function describe () As String
cs	string describe ()
java	String describe ()
py	def describe ()

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.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 reference frame (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

refframe→get_3DCalibrationHint()
refframe→3DCalibrationHint()**YRefFrame**

Returns instructions to proceed to the tridimensional calibration initiated with method start3DCalibration.

```
js function get_3DCalibrationHint( )  
nodejs function get_3DCalibrationHint( )  
php function get_3DCalibrationHint( )  
cpp string get_3DCalibrationHint( )  
m -(NSString*) 3DCalibrationHint  
pas function get_3DCalibrationHint( ): string  
vb function get_3DCalibrationHint( ) As String  
cs string get_3DCalibrationHint( )  
java String get_3DCalibrationHint( )  
py def get_3DCalibrationHint( )  
cmd YRefFrame target get_3DCalibrationHint
```

Returns :

a character string.

refframe→get_3DCalibrationLogMsg()**YRefFrame****refframe→3DCalibrationLogMsg()**

Returns the latest log message from the calibration process.

```
js    function get_3DCalibrationLogMsg( )  
nodejs function get_3DCalibrationLogMsg( )  
php   function get_3DCalibrationLogMsg( )  
cpp   string get_3DCalibrationLogMsg( )  
m     -(NSString*) 3DCalibrationLogMsg  
pas   function get_3DCalibrationLogMsg( ): string  
vb    function get_3DCalibrationLogMsg( ) As String  
cs    string get_3DCalibrationLogMsg( )  
java  String get_3DCalibrationLogMsg( )  
py    def get_3DCalibrationLogMsg( )  
cmd   YRefFrame target get_3DCalibrationLogMsg
```

When no new message is available, returns an empty string.

Returns :

a character string.

refframe→get_3DCalibrationProgress()
refframe→3DCalibrationProgress()**YRefFrame**

Returns the global process indicator for the tridimensional calibration initiated with method start3DCalibration.

```
js function get_3DCalibrationProgress( )  
nodejs function get_3DCalibrationProgress( )  
php function get_3DCalibrationProgress( )  
cpp int get_3DCalibrationProgress( )  
m -(int) 3DCalibrationProgress  
pas function get_3DCalibrationProgress( ): LongInt  
vb function get_3DCalibrationProgress( ) As Integer  
cs int get_3DCalibrationProgress( )  
java int get_3DCalibrationProgress( )  
py def get_3DCalibrationProgress( )  
cmd YRefFrame target get_3DCalibrationProgress
```

Returns :

an integer between 0 (not started) and 100 (stage completed).

refframe→get_3DCalibrationStage()
refframe→3DCalibrationStage()**YRefFrame**

Returns index of the current stage of the calibration initiated with method `start3DCalibration`.

<code>js</code>	<code>function get_3DCalibrationStage()</code>
<code>nodejs</code>	<code>function get_3DCalibrationStage()</code>
<code>php</code>	<code>function get_3DCalibrationStage()</code>
<code>cpp</code>	<code>int get_3DCalibrationStage()</code>
<code>m</code>	<code>-(int) 3DCalibrationStage</code>
<code>pas</code>	<code>function get_3DCalibrationStage(): LongInt</code>
<code>vb</code>	<code>function get_3DCalibrationStage() As Integer</code>
<code>cs</code>	<code>int get_3DCalibrationStage()</code>
<code>java</code>	<code>int get_3DCalibrationStage()</code>
<code>py</code>	<code>def get_3DCalibrationStage()</code>
<code>cmd</code>	<code>YRefFrame target get_3DCalibrationStage</code>

Returns :

an integer, growing each time a calibration stage is completed.

refframe→get_3DCalibrationStageProgress()**YRefFrame****refframe→3DCalibrationStageProgress()**

Returns the process indicator for the current stage of the calibration initiated with method start3DCalibration.

```
js function get_3DCalibrationStageProgress( )  
nodejs function get_3DCalibrationStageProgress( )  
php function get_3DCalibrationStageProgress( )  
cpp int get_3DCalibrationStageProgress( )  
m -(int) 3DCalibrationStageProgress  
pas function get_3DCalibrationStageProgress( ): LongInt  
vb function get_3DCalibrationStageProgress( ) As Integer  
cs int get_3DCalibrationStageProgress( )  
java int get_3DCalibrationStageProgress( )  
py def get_3DCalibrationStageProgress( )  
cmd YRefFrame target get_3DCalibrationStageProgress
```

Returns :

an integer between 0 (not started) and 100 (stage completed).

refframe→get_advertisedValue()
refframe→advertisedValue()**YRefFrame**

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

js	function get_advertisedValue()
nodejs	function get_advertisedValue()
php	function get_advertisedValue()
cpp	string get_advertisedValue()
m	-(NSString*) advertisedValue
pas	function get_advertisedValue() : string
vb	function get_advertisedValue() As String
cs	string get_advertisedValue()
java	String get_advertisedValue()
py	def get_advertisedValue()
cmd	YRefFrame target get_advertisedValue

Returns :

a string corresponding to the current value of the reference frame (no more than 6 characters).

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

refframe→get_bearing()
refframe→bearing()**YRefFrame**

Returns the reference bearing used by the compass.

```
js function get_bearing( )
node.js function get_bearing( )
php function get_bearing( )
cpp double get_bearing( )
m -(double) bearing
pas function get_bearing( ): double
vb function get_bearing( ) As Double
cs double get_bearing( )
java double get_bearing( )
py def get_bearing( )
cmd YRefFrame target get_bearing
```

The relative bearing indicated by the compass is the difference between the measured magnetic heading and the reference bearing indicated here.

Returns :

a floating point number corresponding to the reference bearing used by the compass

On failure, throws an exception or returns Y_BEARING_INVALID.

refframe→getErrorMessage()
refframe→errorMessage()**YRefFrame**

Returns the error message of the latest error with the reference frame.

js	function getErrorMessage()
node.js	function getErrorMessage()
php	function getErrorMessage()
cpp	string getErrorMessage()
m	-(NSString*) errorMessage
pas	function getErrorMessage() : string
vb	function getErrorMessage() As String
cs	string getErrorMessage()
java	String getErrorMessage()
py	def getErrorMessage()

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

Returns :

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

refframe→get_errorType()
refframe→errorType()**YRefFrame**

Returns the numerical error code of the latest error with the reference frame.

js	function get_errorType()
node.js	function get_errorType()
php	function get_errorType()
cpp	YRETCODE get_errorType()
pas	function get_errorType() : YRETCODE
vb	function get_errorType() As YRETCODE
cs	YRETCODE get_errorType()
java	int get_errorType()
py	def get_errorType()

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

Returns :

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

refframe→get_friendlyName()

YRefFrame

Returns a global identifier of the reference frame in the format MODULE_NAME . FUNCTION_NAME.

js	<code>function get_friendlyName()</code>
nodejs	<code>function get_friendlyName()</code>
php	<code>function get_friendlyName()</code>
cpp	<code>string get_friendlyName()</code>
m	<code>-(NSString*) friendlyName</code>
cs	<code>string get_friendlyName()</code>
java	<code>String get_friendlyName()</code>
py	<code>def get_friendlyName()</code>

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

Returns :

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

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

refframe→get_functionDescriptor()
refframe→functionDescriptor()
YRefFrame

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

js	function get_functionDescriptor()
node.js	function get_functionDescriptor()
php	function get_functionDescriptor()
cpp	YFUN_DESCR get_functionDescriptor()
m	-(YFUN_DESCR) functionDescriptor
pas	function get_functionDescriptor() : YFUN_DESCR
vb	function get_functionDescriptor() As YFUN_DESCR
cs	YFUN_DESCR get_functionDescriptor()
java	String get_functionDescriptor()
py	def get_functionDescriptor()

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

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

refframe→get_functionId()
refframe→functionId()**YRefFrame**

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

js	function get_functionId()
node.js	function get_functionId()
php	function get_functionId()
cpp	string get_functionId()
m	-(NSString*) functionId
vb	function get_functionId() As String
cs	string get_functionId()
java	String get_functionId()
py	def get_functionId()

For example relay1

Returns :

a string that identifies the reference frame (ex: relay1)

On failure, throws an exception or returns Y_FUNCTIONID_INVALID.

refframe→get_hardwareId()
refframe→hardwareId()**YRefFrame**

Returns the unique hardware identifier of the reference frame in the form SERIAL.FUNCTIONID.

js	function get_hardwareId()
node.js	function get_hardwareId()
php	function get_hardwareId()
cpp	string get_hardwareId()
m	-(NSString*) hardwareId
vb	function get_hardwareId() As String
cs	string get_hardwareId()
java	String get_hardwareId()
py	def get_hardwareId()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the reference frame (for example RELAYL01-123456.relay1).

Returns :

a string that uniquely identifies the reference frame (ex: RELAYL01-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

refframe→get_logicalName()
refframe→logicalName()**YRefFrame**

Returns the logical name of the reference frame.

js	function get_logicalName()
nodejs	function get_logicalName()
php	function get_logicalName()
cpp	string get_logicalName()
m	-(NSString*) logicalName
pas	function get_logicalName() : string
vb	function get_logicalName() As String
cs	string get_logicalName()
java	String get_logicalName()
py	def get_logicalName()
cmd	YRefFrame target get_logicalName

Returns :

a string corresponding to the logical name of the reference frame.

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

refframe→get_module()
refframe→module()**YRefFrame**

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

```
js   function get_module( )
node.js function get_module( )
php  function get_module( )
cpp   YModule * get_module( )
m    -(YModule*) module
pas   function get_module( ): TYModule
vb    function get_module( ) As YModule
cs   YModule get_module( )
java  YModule get_module( )
py    def get_module( )
```

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

Returns :

an instance of `YModule`

refframe→get_module_async()**YRefFrame****refframe→module_async()**

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

js	function get_module_async(callback, context)
nodejs	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 Javascript 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.

reframe→get_mountOrientation()**YRefFrame****reframe→mountOrientation()**

Returns the installation orientation of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

<code>js</code>	<code>function get_mountOrientation()</code>
<code>nodejs</code>	<code>function get_mountOrientation()</code>
<code>php</code>	<code>function get_mountOrientation()</code>
<code>cpp</code>	<code>Y_MOUNTORIENTATION get_mountOrientation()</code>
<code>m</code>	<code>-(Y_MOUNTORIENTATION) mountOrientation</code>
<code>pas</code>	<code>function get_mountOrientation(): TYMOUNTORIENTATION</code>
<code>vb</code>	<code>function get_mountOrientation() As Y_MOUNTORIENTATION</code>
<code>cs</code>	<code>MOUNTORIENTATION get_mountOrientation()</code>
<code>java</code>	<code>MOUNTORIENTATION get_mountOrientation()</code>
<code>py</code>	<code>def get_mountOrientation()</code>
<code>cmd</code>	<code>YRefFrame target get_mountOrientation</code>

Returns :

a value among the enumeration `Y_MOUNTORIENTATION` (`Y_MOUNTORIENTATION_TWELVE`, `Y_MOUNTORIENTATION_THREE`, `Y_MOUNTORIENTATION_SIX`, `Y_MOUNTORIENTATION_NINE`) corresponding to the orientation of the "X" arrow on the device, as on a clock dial seen from an observer in the center of the box. On the bottom face, the 12H orientation points to the front, while on the top face, the 12H orientation points to the rear.

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

refframe→get_mountPosition()

refframe→mountPosition()

YRefFrame

Returns the installation position of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

js	function get_mountPosition()
nodejs	function get_mountPosition()
php	function get_mountPosition()
cpp	Y_MOUNTPOSITION get_mountPosition()
m	-(Y_MOUNTPOSITION) mountPosition
pas	function get_mountPosition() : TYMOUNTPOSITION
vb	function get_mountPosition() As Y_MOUNTPOSITION
cs	MOUNTPOSITION get_mountPosition()
java	MOUNTPOSITION get_mountPosition()
py	def get_mountPosition()
cmd	YRefFrame target get_mountPosition

Returns :

a value among the **Y_MOUNTPOSITION** enumeration (**Y_MOUNTPOSITION_BOTTOM**, **Y_MOUNTPOSITION_TOP**, **Y_MOUNTPOSITION_FRONT**, **Y_MOUNTPOSITION_RIGHT**, **Y_MOUNTPOSITION_REAR**, **Y_MOUNTPOSITION_LEFT**), corresponding to the installation in a box, on one of the six faces.

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

refframe→get(userData)
refframe→userData()**YRefFrame**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

```
js function get(userData) 
node.js function get(userData) 
php function get(userData) 
cpp void * get(userData) 
m -(void*) userData 
pas function get(userData): Tobject 
vb function get(userData) As Object 
cs object get(userData) 
java Object get(userData) 
py def get(userData)
```

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

Returns :

the object stored previously by the caller.

refframe→isOnline()**YRefFrame**

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

js	function isOnline ()
node.js	function isOnline ()
php	function isOnline ()
cpp	bool isOnline ()
m	-(BOOL) isOnline
pas	function isOnline (): boolean
vb	function isOnline () As Boolean
cs	bool isOnline ()
java	boolean isOnline ()
py	def isOnline ()

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

Returns :

true if the reference frame can be reached, and false otherwise

reframe→isOnline_async()**YRefFrame**

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

```
js   function isOnline_async( callback, context )
nodejs function isOnline_async( callback, context )
```

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

reframe→load()**YRefFrame**

Preloads the reference frame cache with a specified validity duration.

<code>js</code>	<code>function load(msValidity)</code>
<code>node.js</code>	<code>function load(msValidity)</code>
<code>php</code>	<code>function load(\$msValidity)</code>
<code>cpp</code>	<code>YRETCODE load(int msValidity)</code>
<code>m</code>	<code>-(YRETCODE) load : (int) msValidity</code>
<code>pas</code>	<code>function load(msValidity: integer): YRETCODE</code>
<code>vb</code>	<code>function load(ByVal msValidity As Integer) As YRETCODE</code>
<code>cs</code>	<code>YRETCODE load(int msValidity)</code>
<code>java</code>	<code>int load(long msValidity)</code>
<code>py</code>	<code>def load(msValidity)</code>

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.

reframe→load_async()**YRefFrame**

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

```
js   function load_async( msValidity, callback, context)
nodejs 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.

reframe→more3DCalibration()**YRefFrame**

Continues the sensors tridimensional calibration process previously initiated using method `start3DCalibration`.

```
js   function more3DCalibration( )
nodejs function more3DCalibration( )
php  function more3DCalibration( )
cpp   int more3DCalibration( )
m    -(int) more3DCalibration
pas   function more3DCalibration( ): LongInt
vb    function more3DCalibration( ) As Integer
cs   int more3DCalibration( )
java  int more3DCalibration( )
py    def more3DCalibration( )
cmd   YRefFrame target more3DCalibration
```

This method should be called approximately 5 times per second, while positioning the device according to the instructions provided by method `get_3DCalibrationHint`. Note that the instructions change during the calibration process. On failure, throws an exception or returns a negative error code.

refframe→nextRefFrame()**YRefFrame**

Continues the enumeration of reference frames started using `yFirstRefFrame()`.

js	function nextRefFrame()
nodejs	function nextRefFrame()
php	function nextRefFrame()
cpp	YRefFrame * nextRefFrame()
m	-(YRefFrame*) nextRefFrame
pas	function nextRefFrame() : TYRefFrame
vb	function nextRefFrame() As YRefFrame
cs	YRefFrame nextRefFrame()
java	YRefFrame nextRefFrame()
py	def nextRefFrame()

Returns :

a pointer to a `YRefFrame` object, corresponding to a reference frame currently online, or a `null` pointer if there are no more reference frames to enumerate.

reframe→registerValueCallback()**YRefFrame**

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

js	<code>function registerValueCallback(callback)</code>
node.js	<code>function registerValueCallback(callback)</code>
php	<code>function registerValueCallback(\$callback)</code>
cpp	<code>int registerValueCallback(YRefFrameValueCallback callback)</code>
m	<code>-(int) registerValueCallback : (YRefFrameValueCallback) callback</code>
pas	<code>function registerValueCallback(callback: TYRefFrameValueCallback): LongInt</code>
vb	<code>function registerValueCallback() As Integer</code>
cs	<code>int registerValueCallback(ValueCallback callback)</code>
java	<code>int registerValueCallback(UpdateCallback callback)</code>
py	<code>def registerValueCallback(callback)</code>

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.

reframe→save3DCalibration()**YRefFrame**

Applies the sensors tridimensional calibration parameters that have just been computed.

```
js function save3DCalibration( )
nodejs function save3DCalibration( )
php function save3DCalibration( )
cpp int save3DCalibration( )
m -(int) save3DCalibration
pas function save3DCalibration( ): LongInt
vb function save3DCalibration( ) As Integer
cs int save3DCalibration( )
java int save3DCalibration( )
py def save3DCalibration( )
cmd YRefFrame target save3DCalibration
```

Remember to call the `saveToFlash()` method of the module if the changes must be kept when the device is restarted. On failure, throws an exception or returns a negative error code.

refframe→set_bearing()**YRefFrame****refframe→setBearing()**

Changes the reference bearing used by the compass.

<code>js</code>	<code>function set_bearing(newval)</code>
<code>nodejs</code>	<code>function set_bearing(newval)</code>
<code>php</code>	<code>function set_bearing(\$newval)</code>
<code>cpp</code>	<code>int set_bearing(double newval)</code>
<code>m</code>	<code>-(int) setBearing : (double) newval</code>
<code>pas</code>	<code>function set_bearing(newval: double): integer</code>
<code>vb</code>	<code>function set_bearing(ByVal newval As Double) As Integer</code>
<code>cs</code>	<code>int set_bearing(double newval)</code>
<code>java</code>	<code>int set_bearing(double newval)</code>
<code>py</code>	<code>def set_bearing(newval)</code>
<code>cmd</code>	<code>YRefFrame target set_bearing newval</code>

The relative bearing indicated by the compass is the difference between the measured magnetic heading and the reference bearing indicated here. For instance, if you setup as reference bearing the value of the earth magnetic declination, the compass will provide the orientation relative to the geographic North. Similarly, when the sensor is not mounted along the standard directions because it has an additional yaw angle, you can set this angle in the reference bearing so that the compass provides the expected natural direction. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a floating point number corresponding to the reference bearing used by the compass

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

refframe→set_logicalName() refframe→setLogicalName()

YRefFrame

Changes the logical name of the reference frame.

```

js   function set_logicalName( newval)
node.js function set_logicalName( newval)
php  function set_logicalName( $newval)
cpp   int set_logicalName( const string& newval)
m    -(int) setLogicalName : (NSString*) newval
pas   function set_logicalName( newval: string): integer
vb    function set_logicalName( ByVal newval As String) As Integer
cs    int set_logicalName( string newval)
java  int set_logicalName( String newval)
py    def set_logicalName( newval)
cmd   YRefFrame target set_logicalName newval

```

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

Parameters :

newval a string corresponding to the logical name of the reference frame.

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

refframe→set_mountPosition()**YRefFrame****refframe→setMountPosition()**

Changes the compass and tilt sensor frame of reference.

```

js   function set_mountPosition( position, orientation)
nodejs function set_mountPosition( position, orientation)
php  function set_mountPosition( $position, $orientation)
cpp   int set_mountPosition( Y_MOUNTPOSITION position,
                           Y_MOUNTORIENTATION orientation)
m    -(int) setMountPosition : (Y_MOUNTPOSITION) position
                           : (Y_MOUNTORIENTATION) orientation
pas   function set_mountPosition( position: TYMOUNTPOSITION,
                                 orientation: TYMOUNTORIENTATION): LongInt
vb    function set_mountPosition( ) As Integer
cs    int set_mountPosition( MOUNTPOSITION position,
                           MOUNTORIENTATION orientation)
java  int set_mountPosition( MOUNTPOSITION position,
                           MOUNTORIENTATION orientation)
py    def set_mountPosition( position, orientation)
cmd   YRefFrame target set_mountPosition position orientation

```

The magnetic compass and the tilt sensors (pitch and roll) naturally work in the plane parallel to the earth surface. In case the device is not installed upright and horizontally, you must select its reference orientation (parallel to the earth surface) so that the measures are made relative to this position.

Parameters :

position a value among the Y_MOUNTPOSITION enumeration (Y_MOUNTPOSITION_BOTTOM, Y_MOUNTPOSITION_TOP, Y_MOUNTPOSITION_FRONT, Y_MOUNTPOSITION_RIGHT, Y_MOUNTPOSITION_REAR, Y_MOUNTPOSITION_LEFT), corresponding to the installation in a box, on one of the six faces.

orientation a value among the enumeration Y_MOUNTORIENTATION (Y_MOUNTORIENTATION_TWELVE, Y_MOUNTORIENTATION_THREE, Y_MOUNTORIENTATION_SIX, Y_MOUNTORIENTATION_NINE) corresponding to the orientation of the "X" arrow on the device, as on a clock dial seen from an observer in the center of the box. On the bottom face, the 12H orientation points to the front, while on the top face, the 12H orientation points to the rear. Remember to call the saveToFlash() method of the module if the modification must be kept.

refframe→set(userData)
refframe→setUserData()**YRefFrame**

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

```
js   function set(userData) {  
node.js function set(userData) {  
php  function set(userData) {  
cpp   void set(userData) {  
m     -(void) setUserData : (void*) userData  
pas   procedure set(userData: Tobject);  
vb    procedure set(userData: ByVal userData As Object);  
cs    void set(userData: object);  
java  void set(userData: Object);  
py    def set(userData):
```

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

Parameters :

userData any kind of object to be stored

refframe→start3DCalibration()**YRefFrame**

Initiates the sensors tridimensional calibration process.

js	function start3DCalibration()
node.js	function start3DCalibration()
php	function start3DCalibration()
cpp	int start3DCalibration()
m	-(int) start3DCalibration
pas	function start3DCalibration(): LongInt
vb	function start3DCalibration() As Integer
cs	int start3DCalibration()
java	int start3DCalibration()
py	def start3DCalibration()
cmd	YRefFrame target start3DCalibration

This calibration is used at low level for inertial position estimation and to enhance the precision of the tilt sensors. After calling this method, the device should be moved according to the instructions provided by method `get_3DCalibrationHint`, and `more3DCalibration` should be invoked about 5 times per second. The calibration procedure is completed when the method `get_3DCalibrationProgress` returns 100. At this point, the computed calibration parameters can be applied using method `save3DCalibration`. The calibration process can be canceled at any time using method `cancel3DCalibration`. On failure, throws an exception or returns a negative error code.

reframe→wait_async()**YRefFrame**

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

```
js  function wait_async( callback, context)
nodejs function 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.

20.6. Tilt function interface

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

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

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

Global functions

yFindTilt(func)

Retrieves a tilt sensor for a given identifier.

yFirstTilt()

Starts the enumeration of tilt sensors currently accessible.

YTilt methods

tilt→calibrateFromPoints(rawValues, refValues)

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

tilt→describe()

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

tilt→get_advertisedValue()

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

tilt→get_currentRawValue()

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

tilt→get_currentValue()

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

tilt→get_errorMessage()

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

tilt→get_errorType()

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

tilt→get_friendlyName()

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

tilt→get_functionDescriptor()

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

tilt→get_functionId()

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

tilt→get_hardwareId()

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

tilt→get_highestValue()	Returns the maximal value observed for the inclination since the device was started.
tilt→get_logFrequency()	Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.
tilt→get_logicalName()	Returns the logical name of the tilt sensor.
tilt→get_lowestValue()	Returns the minimal value observed for the inclination since the device was started.
tilt→get_module()	Gets the YModule object for the device on which the function is located.
tilt→get_module_async(callback, context)	Gets the YModule object for the device on which the function is located (asynchronous version).
tilt→get_recordedData(startTime, endTime)	Retrieves a DataSet object holding historical data for this sensor, for a specified time interval.
tilt→get_reportFrequency()	Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.
tilt→get_resolution()	Returns the resolution of the measured values.
tilt→get_unit()	Returns the measuring unit for the inclination.
tilt→get(userData)	Returns the value of the userData attribute, as previously stored using method set(userData).
tilt→isOnline()	Checks if the tilt sensor is currently reachable, without raising any error.
tilt→isOnline_async(callback, context)	Checks if the tilt sensor is currently reachable, without raising any error (asynchronous version).
tilt→load(msValidity)	Preloads the tilt sensor cache with a specified validity duration.
tilt→loadCalibrationPoints(rawValues, refValues)	Retrieves error correction data points previously entered using the method calibrateFromPoints.
tilt→load_async(msValidity, callback, context)	Preloads the tilt sensor cache with a specified validity duration (asynchronous version).
tilt→nextTilt()	Continues the enumeration of tilt sensors started using yFirstTilt().
tilt→registerTimedReportCallback(callback)	Registers the callback function that is invoked on every periodic timed notification.
tilt→registerValueCallback(callback)	Registers the callback function that is invoked on every change of advertised value.
tilt→set_highestValue(newval)	Changes the recorded maximal value observed.
tilt→set_logFrequency(newval)	Changes the datalogger recording frequency for this function.
tilt→set_logicalName(newval)	Changes the logical name of the tilt sensor.

tilt→set_lowestValue(newval)

Changes the recorded minimal value observed.

tilt→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

tilt→set_resolution(newval)

Changes the resolution of the measured physical values.

tilt→set_userData(data)

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

tilt→wait_async(callback, context)

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

YTilt.FindTilt() yFindTilt()

YTilt

Retrieves a tilt sensor for a given identifier.

js	<code>function yFindTilt(func)</code>
node.js	<code>function FindTilt(func)</code>
php	<code>function yFindTilt(\$func)</code>
cpp	<code>YTilt* yFindTilt(const string& func)</code>
m	<code>+YTilt*) FindTilt :(NSString*) func</code>
pas	<code>function yFindTilt(func: string): TYTilt</code>
vb	<code>function yFindTilt(ByVal func As String) As YTilt</code>
cs	<code>YTilt FindTilt(string func)</code>
java	<code>YTilt FindTilt(String func)</code>
py	<code>def FindTilt(func)</code>

The identifier can be specified using several formats:

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

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

Parameters :

`func` a string that uniquely characterizes the tilt sensor

Returns :

a `YTilt` object allowing you to drive the tilt sensor.

YTilt.FirstTilt() yFirstTilt()

YTilt

Starts the enumeration of tilt sensors currently accessible.

```
js function yFirstTilt( )
nodejs function FirstTilt( )
php function yFirstTilt( )
cpp YTilt* yFirstTilt( )
m +(YTilt*) FirstTilt
pas function yFirstTilt( ): TYTilt
vb function yFirstTilt( ) As YTilt
cs YTilt FirstTilt( )
java YTilt FirstTilt( )
py def FirstTilt( )
```

Use the method `YTilt.nextTilt()` to iterate on next tilt sensors.

Returns :

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

tilt→calibrateFromPoints()**YTilt**

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)
nodejs function calibrateFromPoints( rawValues, refValues)
php  function calibrateFromPoints( $rawValues, $refValues)
cpp   int calibrateFromPoints( vector<double> rawValues,
                           vector<double> refValues)

m   -(int) calibrateFromPoints : (NSMutableArray*) rawValues
                           : (NSMutableArray*) refValues

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

vb   procedure calibrateFromPoints( )

cs   int calibrateFromPoints( List<double> rawValues,
                           List<double> refValues)

java int calibrateFromPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)

py   def calibrateFromPoints( rawValues, refValues)
cmd  YTilt target calibrateFromPoints rawValues refValues

```

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

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

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

tilt→describe()**YTilt**

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

js	function describe ()
node.js	function describe ()
php	function describe ()
cpp	string describe ()
m	-(NSString*) describe
pas	function describe (): string
vb	function describe () As String
cs	string describe ()
java	String describe ()
py	def describe ()

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the tilt sensor (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

tilt→get_advertisedValue()
tilt→advertisedValue()

YTilt

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

```
js function get_advertisedValue( )
node.js function get_advertisedValue( )
php function get_advertisedValue( )
cpp string get_advertisedValue( )
m -(NSString*) advertisedValue
pas function get_advertisedValue( ): string
vb function get_advertisedValue( ) As String
cs string get_advertisedValue( )
java String get_advertisedValue( )
py def get_advertisedValue( )
cmd YTilt target get_advertisedValue
```

Returns :

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

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

tilt→get_currentRawValue() tilt→currentRawValue()

YTilt

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

```
js function get_currentRawValue( )
nodejs function get_currentRawValue( )
php function get_currentRawValue( )
cpp double get_currentRawValue( )
m -(double) currentRawValue
pas function get_currentRawValue( ): double
vb function get_currentRawValue( ) As Double
cs double get_currentRawValue( )
java double get_currentRawValue( )
py def get_currentRawValue( )
cmd YTilt target get_currentRawValue
```

Returns :

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

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

tilt→get_currentValue()
tilt→currentValue()**YTilt**

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

```
js function get_currentValue( )
node.js function get_currentValue( )
php function get_currentValue( )
cpp double get_currentValue( )
m -(double) currentValue
pas function get_currentValue( ): double
vb function get_currentValue( ) As Double
cs double get_currentValue( )
java double get_currentValue( )
py def get_currentValue( )
cmd YTilt target get_currentValue
```

Returns :

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

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

tilt→get_errorMessage() tilt→errorMessage()

YTilt

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

```
js   function get_errorMessage( )  
nodejs function get_errorMessage( )  
php  function get_errorMessage( )  
cpp   string get_errorMessage( )  
m    -(NSString*) errorMessage  
pas   function get_errorMessage( ): string  
vb    function get_errorMessage( ) As String  
cs    string get_errorMessage( )  
java  String get_errorMessage( )  
py    def get_errorMessage( )
```

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

Returns :

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

**tilt→get_errorType()
tilt→errorType()****YTilt**

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

```
js   function get_errorType( )  
node.js function get_errorType( )  
php  function get_errorType( )  
cpp   YRETCODE get_errorType( )  
pas   function get_errorType( ): YRETCODE  
vb    function get_errorType( ) As YRETCODE  
cs    YRETCODE get_errorType( )  
java  int get_errorType( )  
py    def get_errorType( )
```

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

Returns :

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

tilt→get_friendlyName() tilt→friendlyName()

YTilt

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

js	<code>function get_friendlyName()</code>
nodejs	<code>function get_friendlyName()</code>
php	<code>function get_friendlyName()</code>
cpp	<code>string get_friendlyName()</code>
m	<code>-(NSString*) friendlyName</code>
cs	<code>string get_friendlyName()</code>
java	<code>String get_friendlyName()</code>
py	<code>def get_friendlyName()</code>

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

Returns :

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

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

tilt→get_functionDescriptor()
tilt→functionDescriptor()**YTilt**

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

```
js    function get_functionDescriptor( )  
node.js function get_functionDescriptor( )  
php   function get_functionDescriptor( )  
cpp   YFUN_DESCR get_functionDescriptor( )  
m     -(YFUN_DESCR) functionDescriptor  
pas   function get_functionDescriptor( ): YFUN_DESCR  
vb    function get_functionDescriptor( ) As YFUN_DESCR  
cs    YFUN_DESCR get_functionDescriptor( )  
java  String get_functionDescriptor( )  
py    def get_functionDescriptor( )
```

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

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

tilt→get_functionId() tilt→functionId()

YTilt

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

```
js function get_functionId( )
node.js function get_functionId( )
php function get_functionId( )
cpp string get_functionId( )
m -(NSString*) functionId
vb function get_functionId( ) As String
cs string get_functionId( )
java String get_functionId( )
py def get_functionId( )
```

For example relay1

Returns :

a string that identifies the tilt sensor (ex: relay1)

On failure, throws an exception or returns Y_FUNCTIONID_INVALID.

tilt→get_hardwareId()**YTilt**

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

```
js function get_hardwareId( )  
node.js function get_hardwareId( )  
php function get_hardwareId( )  
cpp string get_hardwareId( )  
m -(NSString*) hardwareId  
vb function get_hardwareId( ) As String  
cs string get_hardwareId( )  
java String get_hardwareId( )  
py def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the tilt sensor (for example RELAYL01-123456.relay1).

Returns :

a string that uniquely identifies the tilt sensor (ex: RELAYL01-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

tilt→get_highestValue() tilt→highestValue()

YTilt

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

```
js   function get_highestValue( )  
node.js function get_highestValue( )  
php  function get_highestValue( )  
cpp   double get_highestValue( )  
m    -(double) highestValue  
pas   function get_highestValue( ): double  
vb    function get_highestValue( ) As Double  
cs    double get_highestValue( )  
java  double get_highestValue( )  
py    def get_highestValue( )  
cmd   YTilt target get_highestValue
```

Returns :

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

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

tilt→get_logFrequency()
tilt→logFrequency()**YTilt**

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( )  
nodejs function get_logFrequency( )  
php  function get_logFrequency( )  
cpp   string get_logFrequency( )  
m    -(NSString*) logFrequency  
pas   function get_logFrequency( ):string  
vb    function get_logFrequency( ) As String  
cs    string get_logFrequency( )  
java  String get_logFrequency( )  
py    def get_logFrequency( )  
cmd  YTilt target get_logFrequency
```

Returns :

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

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

tilt→get_logicalName()

YTilt

tilt→logicalName()

Returns the logical name of the tilt sensor.

js	function get_logicalName()
nodejs	function get_logicalName()
php	function get_logicalName()
cpp	string get_logicalName()
m	-(NSString*) logicalName
pas	function get_logicalName() : string
vb	function get_logicalName() As String
cs	string get_logicalName()
java	String get_logicalName()
py	def get_logicalName()
cmd	YTilt target get_logicalName

Returns :

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

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

tilt→get_lowestValue()**YTilt**

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

js function **get_lowestValue()****node.js** function **get_lowestValue()****php** function **get_lowestValue()****cpp** double **get_lowestValue()****m** -(double) lowestValue**pas** function **get_lowestValue()**: double**vb** function **get_lowestValue()** As Double**cs** double **get_lowestValue()****java** double **get_lowestValue()****py** def **get_lowestValue()****cmd** **YTilt target get_lowestValue****Returns :**

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

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

tilt→get_module()**YTilt****tilt→module()**

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

js	function get_module()
nodejs	function get_module()
php	function get_module()
cpp	YModule * get_module()
m	-(YModule*) module
pas	function get_module() : TYModule
vb	function get_module() As YModule
cs	YModule get_module()
java	YModule get_module()
py	def get_module()

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

Returns :

an instance of YModule

tilt→get_module_async()
tilt→module_async()**YTilt**

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

```
js  function get_module_async( callback, context )
node.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 Javascript 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.

tilt→get_recordedData() tilt→recordedData()

YTilt

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

```

js   function get_recordedData( startTime, endTime)
nodejs function get_recordedData( startTime, endTime)
php  function get_recordedData( $startTime, $endTime)
cpp   YDataSet get_recordedData( s64 startTime, s64 endTime)
m     -(YDataSet*) recordedData : (s64) startTime
                  : (s64) endTime

pas   function get_recordedData( startTime: int64, endTime: int64): TYDataSet
vb    function get_recordedData( ) As YDataSet
cs    YDataSet get_recordedData( long startTime, long endTime)
java   YDataSet get_recordedData( long startTime, long endTime)
py    def get_recordedData( startTime, endTime)
cmd   YTilt target get_recordedData startTime endTime

```

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

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

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any 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.

tilt→get_reportFrequency()
tilt→reportFrequency()**YTilt**

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

```
js  function get_reportFrequency( )  
nodejs function get_reportFrequency( )  
php  function get_reportFrequency( )  
cpp   string get_reportFrequency( )  
m    -(NSString*) reportFrequency  
pas   function get_reportFrequency( ): string  
vb    function get_reportFrequency( ) As String  
cs    string get_reportFrequency( )  
java  String get_reportFrequency( )  
py    def get_reportFrequency( )  
cmd   YTilt target get_reportFrequency
```

Returns :

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

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

tilt→get_resolution() tilt→resolution()

YTilt

Returns the resolution of the measured values.

```
js function get_resolution( )
nodejs function get_resolution( )
php function get_resolution( )
cpp double get_resolution( )
m -(double) resolution
pas function get_resolution( ): double
vb function get_resolution( ) As Double
cs double get_resolution( )
java double get_resolution( )
py def get_resolution( )
cmd YTilt target get_resolution
```

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

Returns :

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

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

tilt→get_unit()**YTilt****tilt→unit()**

Returns the measuring unit for the inclination.

```
js function get_unit( )
node.js function get_unit( )
php function get_unit( )
cpp string get_unit( )
m -(NSString*) unit
pas function get_unit( ): string
vb function get_unit( ) As String
cs string get_unit( )
java String get_unit( )
py def get_unit( )
cmd YTilt target get_unit
```

Returns :

a string corresponding to the measuring unit for the inclination

On failure, throws an exception or returns Y_UNIT_INVALID.

tilt→get(userData)

YTilt

tilt→userData()

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

js	<code>function get(userData) </code>
nodejs	<code>function get(userData) </code>
php	<code>function get(userData) </code>
cpp	<code>void * get(userData) </code>
m	<code>-(void*) userData</code>
pas	<code>function get(userData): Tobject</code>
vb	<code>function get(userData) As Object</code>
cs	<code>object get(userData) </code>
java	<code>Object get(userData) </code>
py	<code>def get(userData) </code>

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

Returns :

the object stored previously by the caller.

tilt→isOnline()

YTilt

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

js	function isOnline()
nodejs	function isOnline()
php	function isOnline()
cpp	bool isOnline()
m	- (BOOL) isOnline
pas	function isOnline() : boolean
vb	function isOnline() As Boolean
cs	bool isOnline()
java	boolean isOnline()
py	def isOnline()

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

Returns :

true if the tilt sensor can be reached, and false otherwise

tilt→isOnline_async()

YTilt

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

```
js   function isOnline_async( callback, context)
nodejs function isOnline_async( callback, context)
```

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

tilt→load()

YTilt

Preloads the tilt sensor cache with a specified validity duration.

<code>js</code>	<code>function load(msValidity)</code>
<code>nodejs</code>	<code>function load(msValidity)</code>
<code>php</code>	<code>function load(\$msValidity)</code>
<code>cpp</code>	<code>YRETCODE load(int msValidity)</code>
<code>m</code>	<code>-(YRETCODE) load : (int) msValidity</code>
<code>pas</code>	<code>function load(msValidity: integer): YRETCODE</code>
<code>vb</code>	<code>function load(ByVal msValidity As Integer) As YRETCODE</code>
<code>cs</code>	<code>YRETCODE load(int msValidity)</code>
<code>java</code>	<code>int load(long msValidity)</code>
<code>py</code>	<code>def load(msValidity)</code>

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

Parameters :

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

tilt→loadCalibrationPoints()**YTilt**

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

```

js   function loadCalibrationPoints( rawValues, refValues)
nodejs function loadCalibrationPoints( rawValues, refValues)
php  function loadCalibrationPoints( &$rawValues, &$refValues)
cpp   int loadCalibrationPoints( vector<double>& rawValues,
                                vector<double>& refValues)

m    -(int) loadCalibrationPoints : (NSMutableArray*) rawValues
                           : (NSMutableArray*) refValues

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

vb   procedure loadCalibrationPoints( )
cs   int loadCalibrationPoints( List<double> rawValues,
                           List<double> refValues)

java int loadCalibrationPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)

py   def loadCalibrationPoints( rawValues, refValues)
cmd  YTilt target loadCalibrationPoints rawValues refValues

```

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

tilt→load_async()

YTilt

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

```
js function load_async( msValidity, callback, context)
nodejs 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.

tilt→nextTilt()**YTilt**

Continues the enumeration of tilt sensors started using `yFirstTilt()`.

<code>js</code>	<code>function nextTilt()</code>
<code>node.js</code>	<code>function nextTilt()</code>
<code>php</code>	<code>function nextTilt()</code>
<code>cpp</code>	<code>YTilt * nextTilt()</code>
<code>m</code>	<code>-(YTilt*) nextTilt</code>
<code>pas</code>	<code>function nextTilt(): TYTilt</code>
<code>vb</code>	<code>function nextTilt() As YTilt</code>
<code>cs</code>	<code>YTilt nextTilt()</code>
<code>java</code>	<code>YTilt nextTilt()</code>
<code>py</code>	<code>def nextTilt()</code>

Returns :

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

tilt→registerTimedReportCallback()

YTilt

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

<code>js</code>	<code>function registerTimedReportCallback(callback)</code>
<code>node.js</code>	<code>function registerTimedReportCallback(callback)</code>
<code>php</code>	<code>function registerTimedReportCallback(\$callback)</code>
<code>cpp</code>	<code>int registerTimedReportCallback(YTiltTimedReportCallback callback)</code>
<code>m</code>	<code>-(int) registerTimedReportCallback : (YTiltTimedReportCallback) callback</code>
<code>pas</code>	<code>function registerTimedReportCallback(callback: YTiltTimedReportCallback): LongInt</code>
<code>vb</code>	<code>function registerTimedReportCallback() As Integer</code>
<code>cs</code>	<code>int registerTimedReportCallback(TimedReportCallback callback)</code>
<code>java</code>	<code>int registerTimedReportCallback(TimedReportCallback callback)</code>
<code>py</code>	<code>def registerTimedReportCallback(callback)</code>

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

Parameters :

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

tilt→registerValueCallback()

YTilt

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

js	<code>function registerValueCallback(callback)</code>
node.js	<code>function registerValueCallback(callback)</code>
php	<code>function registerValueCallback(\$callback)</code>
cpp	<code>int registerValueCallback(YTiltValueCallback callback)</code>
m	<code>-(int) registerValueCallback : (YTiltValueCallback) callback</code>
pas	<code>function registerValueCallback(callback: TYTiltValueCallback): LongInt</code>
vb	<code>function registerValueCallback() As Integer</code>
cs	<code>int registerValueCallback(ValueCallback callback)</code>
java	<code>int registerValueCallback(UpdateCallback callback)</code>
py	<code>def registerValueCallback(callback)</code>

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

Parameters :

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

tilt→set_highestValue()
tilt→setHighestValue()**YTilt**

Changes the recorded maximal value observed.

```
js function set_highestValue( newval)
node.js function set_highestValue( newval)
php function set_highestValue( $newval)
cpp int set_highestValue( double newval)
m -(int) setHighestValue : (double) newval
pas function set_highestValue( newval: double): integer
vb function set_highestValue( ByVal newval As Double) As Integer
cs int set_highestValue( double newval)
java int set_highestValue( double newval)
py def set_highestValue( newval)
cmd YTilt target set_highestValue newval
```

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

tilt→set_logFrequency() tilt→setLogFrequency()

YTilt

Changes the datalogger recording frequency for this function.

<code>js</code>	<code>function set_logFrequency(newval)</code>
<code>nodejs</code>	<code>function set_logFrequency(newval)</code>
<code>php</code>	<code>function set_logFrequency(\$newval)</code>
<code>cpp</code>	<code>int set_logFrequency(const string& newval)</code>
<code>m</code>	<code>-(int) setLogFrequency : (NSString*) newval</code>
<code>pas</code>	<code>function set_logFrequency(newval: string): integer</code>
<code>vb</code>	<code>function set_logFrequency(ByVal newval As String) As Integer</code>
<code>cs</code>	<code>int set_logFrequency(string newval)</code>
<code>java</code>	<code>int set_logFrequency(String newval)</code>
<code>py</code>	<code>def set_logFrequency(newval)</code>
<code>cmd</code>	<code>YTilt target set_logFrequency newval</code>

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

tilt→set_logicalName() tilt→setLogicalName()

YTilt

Changes the logical name of the tilt sensor.

```

js   function set_logicalName( newval)
node.js function set_logicalName( newval)
php  function set_logicalName( $newval)
cpp   int set_logicalName( const string& newval)
m    -(int) setLogicalName : (NSString*) newval
pas   function set_logicalName( newval: string): integer
vb    function set_logicalName( ByVal newval As String) As Integer
cs    int set_logicalName( string newval)
java  int set_logicalName( String newval)
py    def set_logicalName( newval)
cmd   YTilt target set_logicalName newval

```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

tilt→set_lowestValue() tilt→setLowestValue()

YTilt

Changes the recorded minimal value observed.

<code>js</code>	<code>function set_lowestValue(newval)</code>
<code>node.js</code>	<code>function set_lowestValue(newval)</code>
<code>php</code>	<code>function set_lowestValue(\$newval)</code>
<code>cpp</code>	<code>int set_lowestValue(double newval)</code>
<code>m</code>	<code>-(int) setLowestValue : (double) newval</code>
<code>pas</code>	<code>function set_lowestValue(newval: double): integer</code>
<code>vb</code>	<code>function set_lowestValue(ByVal newval As Double) As Integer</code>
<code>cs</code>	<code>int set_lowestValue(double newval)</code>
<code>java</code>	<code>int set_lowestValue(double newval)</code>
<code>py</code>	<code>def set_lowestValue(newval)</code>
<code>cmd</code>	<code>YTilt target set_lowestValue newval</code>

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

tilt→set_reportFrequency() tilt→setReportFrequency()

YTilt

Changes the timed value notification frequency for this function.

```

js   function set_reportFrequency( newval)
node.js function set_reportFrequency( newval)
php  function set_reportFrequency( $newval)
cpp   int set_reportFrequency( const string& newval)
m    -(int) setReportFrequency : (NSString*) newval
pas   function set_reportFrequency( newval: string): integer
vb    function set_reportFrequency( ByVal newval As String) As Integer
cs    int set_reportFrequency( string newval)
java  int set_reportFrequency( String newval)
py    def set_reportFrequency( newval)
cmd   YTilt target set_reportFrequency newval

```

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

tilt→set_resolution() tilt→setResolution()

YTilt

Changes the resolution of the measured physical values.

js	<code>function set_resolution(newval)</code>
node.js	<code>function set_resolution(newval)</code>
php	<code>function set_resolution(\$newval)</code>
cpp	<code>int set_resolution(double newval)</code>
m	<code>-(int) setResolution : (double) newval</code>
pas	<code>function set_resolution(newval: double): integer</code>
vb	<code>function set_resolution(ByVal newval As Double) As Integer</code>
cs	<code>int set_resolution(double newval)</code>
java	<code>int set_resolution(double newval)</code>
py	<code>def set_resolution(newval)</code>
cmd	<code>YTilt target set_resolution newval</code>

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

tilt→set(userData)**YTilt**

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

```
js   function setUserData( data)
node.js function setUserData( data)
php  function setUserData( $data)
cpp   void setUserData( void* data)
m    -(void) setUserData : (void*) data
pas   procedure setUserData( data: Tobject)
vb    procedure setUserData( ByVal data As Object)
cs    void setUserData( object data)
java  void setUserData( Object data)
py    def setUserData( 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

tilt→wait_async()

YTilt

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

```
js  function wait_async( callback, context )
nodejs function 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.

20.7. Compass function interface

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

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

```

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

```

Global functions

yFindCompass(func)

Retrieves a compass for a given identifier.

yFirstCompass()

Starts the enumeration of compasses currently accessible.

YCompass methods

compass→calibrateFromPoints(rawValues, refValues)

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

compass→describe()

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

compass→get_advertisedValue()

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

compass→get_currentRawValue()

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

compass→get_currentValue()

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

compass→get_errorMessage()

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

compass→get_errorType()

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

compass→get_friendlyName()

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

compass→get_functionDescriptor()

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

compass→get_functionId()

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

compass→get_hardwareId()

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

compass→get_highestValue()

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

compass→get_logFrequency()

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

compass→get_logicalName()

Returns the logical name of the compass.

compass→get_lowestValue()

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

compass→get_magneticHeading()

Returns the magnetic heading, regardless of the configured bearing.

compass→get_module()

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

compass→get_module_async(callback, context)

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

compass→get_recordedData(startTime, endTime)

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

compass→get_reportFrequency()

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

compass→get_resolution()

Returns the resolution of the measured values.

compass→get_unit()

Returns the measuring unit for the relative bearing.

compass→get(userData)

Returns the value of the userData attribute, as previously stored using method set(userData).

compass→isOnline()

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

compass→isOnline_async(callback, context)

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

compass→load(msValidity)

Preloads the compass cache with a specified validity duration.

compass→loadCalibrationPoints(rawValues, refValues)

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

compass→load_async(msValidity, callback, context)

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

compass→nextCompass()

Continues the enumeration of compasses started using yFirstCompass().

compass→registerTimedReportCallback(callback)

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

compass→registerValueCallback(callback)

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

compass→set_highestValue(newval)

Changes the recorded maximal value observed.

compass→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

compass→set_logicalName(newval)

Changes the logical name of the compass.

compass→set_lowestValue(newval)

Changes the recorded minimal value observed.

compass→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

compass→set_resolution(newval)

Changes the resolution of the measured physical values.

compass→set(userData)

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

compass→wait_async(callback, context)

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

YCompass.FindCompass() yFindCompass()

YCompass

Retrieves a compass for a given identifier.

<code>js</code>	<code>function yFindCompass(func)</code>
<code>nodejs</code>	<code>function FindCompass(func)</code>
<code>php</code>	<code>function yFindCompass(\$func)</code>
<code>cpp</code>	<code>YCompass* yFindCompass(const string& func)</code>
<code>m</code>	<code>+ (YCompass*) FindCompass :(NSString*) func</code>
<code>pas</code>	<code>function yFindCompass(func: string): TYCompass</code>
<code>vb</code>	<code>function yFindCompass(ByVal func As String) As YCompass</code>
<code>cs</code>	<code>YCompass FindCompass(string func)</code>
<code>java</code>	<code>YCompass FindCompass(String func)</code>
<code>py</code>	<code>def FindCompass(func)</code>

The identifier can be specified using several formats:

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

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

Parameters :

`func` a string that uniquely characterizes the compass

Returns :

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

YCompass.FirstCompass() yFirstCompass()

YCompass

Starts the enumeration of compasses currently accessible.

```
js function yFirstCompass( )
node.js function FirstCompass( )
php function yFirstCompass( )
cpp YCompass* yFirstCompass( )
m +(YCompass*) FirstCompass
pas function yFirstCompass( ): TYCompass
vb function yFirstCompass( ) As YCompass
cs YCompass FirstCompass( )
java YCompass FirstCompass( )
py def FirstCompass( )
```

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

Returns :

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

compass→calibrateFromPoints()**YCompass**

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)
nodejs function calibrateFromPoints( rawValues, refValues)
php  function calibrateFromPoints( $rawValues, $refValues)
cpp   int calibrateFromPoints( vector<double> rawValues,
                               vector<double> refValues)
m    -(int) calibrateFromPoints : (NSMutableArray*) rawValues
                           : (NSMutableArray*) refValues
pas   function calibrateFromPoints( rawValues: TDoubleArray,
                                   refValues: TDoubleArray): LongInt
vb    procedure calibrateFromPoints( )
cs    int calibrateFromPoints( List<double> rawValues,
                           List<double> refValues)
java  int calibrateFromPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)
py    def calibrateFromPoints( rawValues, refValues)
cmd   YCompass target calibrateFromPoints rawValues refValues

```

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

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

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

compass→describe()**YCompass**

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

js	function describe()
nodejs	function describe()
php	function describe()
cpp	string describe()
m	-(NSString*) describe
pas	function describe() : string
vb	function describe() As String
cs	string describe()
java	String describe()
py	def describe()

More precisely, TYPE is the type of the function, NAME is the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the compass (ex: Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

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

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

js	function get_advertisedValue()
nodejs	function get_advertisedValue()
php	function get_advertisedValue()
cpp	string get_advertisedValue()
m	-(NSString*) advertisedValue
pas	function get_advertisedValue() : string
vb	function get_advertisedValue() As String
cs	string get_advertisedValue()
java	String get_advertisedValue()
py	def get_advertisedValue()
cmd	YCompass target get_advertisedValue

Returns :

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

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

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

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

```
js function get_currentRawValue( )
nodejs function get_currentRawValue( )
php function get_currentRawValue( )
cpp double get_currentRawValue( )
m -(double) currentRawValue
pas function get_currentRawValue( ): double
vb function get_currentRawValue( ) As Double
cs double get_currentRawValue( )
java double get_currentRawValue( )
py def get_currentRawValue( )
cmd YCompass target get_currentRawValue
```

Returns :

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

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

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

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

js	function get_currentValue()
node.js	function get_currentValue()
php	function get_currentValue()
cpp	double get_currentValue()
m	-(double) currentValue
pas	function get_currentValue() : double
vb	function get_currentValue() As Double
cs	double get_currentValue()
java	double get_currentValue()
py	def get_currentValue()
cmd	YCompass target get_currentValue

Returns :

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

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

compass→get_errorMessage()
compass→errorMessage()**YCompass**

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

js	function get_errorMessage()
node.js	function get_errorMessage()
php	function get_errorMessage()
cpp	string get_errorMessage()
m	- (NSString*) errorMessage
pas	function get_errorMessage(): string
vb	function get_errorMessage() As String
cs	string get_errorMessage()
java	String getErrorMessage()
py	def getErrorMessage()

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

Returns :

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

compass→get_errorType()
compass→errorType()**YCompass**

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

js	function get_errorType()
node.js	function get_errorType()
php	function get_errorType()
cpp	YRETCODE get_errorType()
pas	function get_errorType() : YRETCODE
vb	function get_errorType() As YRETCODE
cs	YRETCODE get_errorType()
java	int get_errorType()
py	def get_errorType()

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

Returns :

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

compass→get_friendlyName()
compass→friendlyName()**YCompass**

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

```
js   function get_friendlyName( )  
node.js function get_friendlyName( )  
php  function get_friendlyName( )  
cpp   string get_friendlyName( )  
m    -(NSString*) friendlyName  
cs   string get_friendlyName( )  
java  String get_friendlyName( )  
py   def get_friendlyName( )
```

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

Returns :

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

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

compass->get_functionDescriptor() compass->functionDescriptor()

YCompass

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

js	function get_functionDescriptor()
nodejs	function get_functionDescriptor()
php	function get_functionDescriptor()
cpp	YFUN_DESCR get_functionDescriptor()
m	-(YFUN_DESCR) functionDescriptor
pas	function get_functionDescriptor() : YFUN_DESCR
vb	function get_functionDescriptor() As YFUN_DESCR
cs	YFUN_DESCR get_functionDescriptor()
java	String get_functionDescriptor()
py	def get_functionDescriptor()

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

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

**compass→get_functionId()
compass→functionId()****YCompass**

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

js	function get_functionId()
node.js	function get_functionId()
php	function get_functionId()
cpp	string get_functionId()
m	-(NSString*) functionId
vb	function get_functionId() As String
cs	string get_functionId()
java	String get_functionId()
py	def get_functionId()

For example `relay1`

Returns :

a string that identifies the compass (ex: `relay1`)

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

compass→get_hardwareId() compass→hardwareId()

YCompass

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

js	function get_hardwareId()
nodejs	function get_hardwareId()
php	function get_hardwareId()
cpp	string get_hardwareId()
m	-(NSString*) hardwareId
vb	function get_hardwareId() As String
cs	string get_hardwareId()
java	String get_hardwareId()
py	def get_hardwareId()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the compass (for example RELAYL01-123456.relay1).

Returns :

a string that uniquely identifies the compass (ex: RELAYL01-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

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

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

js	function get_highestValue()
node.js	function get_highestValue()
php	function get_highestValue()
cpp	double get_highestValue()
m	-(double) highestValue
pas	function get_highestValue() : double
vb	function get_highestValue() As Double
cs	double get_highestValue()
java	double get_highestValue()
py	def get_highestValue()
cmd	YCompass target get_highestValue

Returns :

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

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

compass→get_logFrequency() compass→logFrequency()

YCompass

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()
nodejs	function get_logFrequency()
php	function get_logFrequency()
cpp	string get_logFrequency()
m	-(NSString*) logFrequency
pas	function get_logFrequency() : string
vb	function get_logFrequency() As String
cs	string get_logFrequency()
java	String get_logFrequency()
py	def get_logFrequency()
cmd	YCompass target get_logFrequency

Returns :

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

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

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

Returns the logical name of the compass.

```
js   function get_logicalName( )  
node.js function get_logicalName( )  
php  function get_logicalName( )  
cpp   string get_logicalName( )  
m    -(NSString*) logicalName  
pas  function get_logicalName( ): string  
vb   function get_logicalName( ) As String  
cs   string get_logicalName( )  
java String get_logicalName( )  
py   def get_logicalName( )  
cmd  YCompass target get_logicalName
```

Returns :

a string corresponding to the logical name of the compass.

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

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

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

js	function get_lowestValue()
node.js	function get_lowestValue()
php	function get_lowestValue()
cpp	double get_lowestValue()
m	-(double) lowestValue
pas	function get_lowestValue(): double
vb	function get_lowestValue() As Double
cs	double get_lowestValue()
java	double get_lowestValue()
py	def get_lowestValue()
cmd	YCompass target get_lowestValue

Returns :

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

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

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

Returns the magnetic heading, regardless of the configured bearing.

```
js   function get_magneticHeading( )  
node.js function get_magneticHeading( )  
php  function get_magneticHeading( )  
cpp   double get_magneticHeading( )  
m    -(double) magneticHeading  
pas   function get_magneticHeading( ): double  
vb    function get_magneticHeading( ) As Double  
cs    double get_magneticHeading( )  
java   double get_magneticHeading( )  
py    def get_magneticHeading( )  
cmd  YCompass target get_magneticHeading
```

Returns :

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

On failure, throws an exception or returns Y_MAGNETICHEADING_INVALID.

compass→get_module()
compass→module()**YCompass**

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

js	function get_module()
nodejs	function get_module()
php	function get_module()
cpp	YModule * get_module()
m	-(YModule*) module
pas	function get_module() : TYModule
vb	function get_module() As YModule
cs	YModule get_module()
java	YModule get_module()
py	def get_module()

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

Returns :

an instance of YModule

compass→get_module_async()
compass→module_async()**YCompass**

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

```
js   function get_module_async( callback, context )
node.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 Javascript 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.

compass→get_recordedData() compass→recordedData()

YCompass

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

js	function get_recordedData(startTime, endTime)
node.js	function get_recordedData(startTime, endTime)
php	function get_recordedData(\$startTime, \$endTime)
cpp	YDataSet get_recordedData(s64 startTime, s64 endTime)
m	-(YDataSet*) recordedData : (s64) startTime : (s64) endTime
pas	function get_recordedData(startTime: int64, endTime: int64): TYDataSet
vb	function get_recordedData() As YDataSet
cs	YDataSet get_recordedData(long startTime, long endTime)
java	YDataSet get_recordedData(long startTime, long endTime)
py	def get_recordedData(startTime, endTime)
cmd	YCompass target get_recordedData startTime endTime

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

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

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any 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.

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

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

js	function get_reportFrequency() ()
nodejs	function get_reportFrequency() ()
php	function get_reportFrequency() ()
cpp	string get_reportFrequency() ()
m	- (NSString*) reportFrequency
pas	function get_reportFrequency() : string
vb	function get_reportFrequency() As String
cs	string get_reportFrequency() ()
java	String get_reportFrequency() ()
py	def get_reportFrequency() ()
cmd	YCompass target get_reportFrequency

Returns :

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

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

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

Returns the resolution of the measured values.

js	function get_resolution()
node.js	function get_resolution()
php	function get_resolution()
cpp	double get_resolution()
m	-(double) resolution
pas	function get_resolution(): double
vb	function get_resolution() As Double
cs	double get_resolution()
java	double get_resolution()
py	def get_resolution()
cmd	YCompass target get_resolution

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

Returns :

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

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

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

Returns the measuring unit for the relative bearing.

js	function get_unit()
node.js	function get_unit()
php	function get_unit()
cpp	string get_unit()
m	-(NSString*) unit
pas	function get_unit() : string
vb	function get_unit() As String
cs	string get_unit()
java	String get_unit()
py	def get_unit()
cmd	YCompass target get_unit

Returns :

a string corresponding to the measuring unit for the relative bearing

On failure, throws an exception or returns Y_UNIT_INVALID.

compass→get(userData)**YCompass****compass→userData()**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

js	<code>function get(userData) </code>
nodejs	<code>function get(userData) </code>
php	<code>function get(userData) </code>
cpp	<code>void * get(userData) </code>
m	<code>-(void*) userData</code>
pas	<code>function get(userData): Tobject</code>
vb	<code>function get(userData) As Object</code>
cs	<code>object get(userData) </code>
java	<code>Object get(userData) </code>
py	<code>def get(userData) </code>

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

Returns :

the object stored previously by the caller.

compass→isOnline()

YCompass

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

js	function isOnline ()
nodejs	function isOnline ()
php	function isOnline ()
cpp	bool isOnline ()
m	- (BOOL) isOnline
pas	function isOnline (): boolean
vb	function isOnline () As Boolean
cs	bool isOnline ()
java	boolean isOnline ()
py	def isOnline ()

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

Returns :

true if the compass can be reached, and false otherwise

compass→isOnline_async()**YCompass**

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

js	function isOnline_async(callback, context)
node.js	function isOnline_async(callback, context)

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

compass→load()**YCompass**

Preloads the compass cache with a specified validity duration.

<code>js</code>	<code>function load(msValidity)</code>
<code>nodejs</code>	<code>function load(msValidity)</code>
<code>php</code>	<code>function load(\$msValidity)</code>
<code>cpp</code>	<code>YRETCODE load(int msValidity)</code>
<code>m</code>	<code>-(YRETCODE) load : (int) msValidity</code>
<code>pas</code>	<code>function load(msValidity: integer): YRETCODE</code>
<code>vb</code>	<code>function load(ByVal msValidity As Integer) As YRETCODE</code>
<code>cs</code>	<code>YRETCODE load(int msValidity)</code>
<code>java</code>	<code>int load(long msValidity)</code>
<code>py</code>	<code>def load(msValidity)</code>

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

Parameters :

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

compass→loadCalibrationPoints()**YCompass**

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

```

js   function loadCalibrationPoints( rawValues, refValues)
nodejs function loadCalibrationPoints( rawValues, refValues)
php  function loadCalibrationPoints( &$rawValues, &$refValues)
cpp   int loadCalibrationPoints( vector<double>& rawValues,
                                vector<double>& refValues)

m    -(int) loadCalibrationPoints : (NSMutableArray*) rawValues
                           : (NSMutableArray*) refValues

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

vb   procedure loadCalibrationPoints( )
cs   int loadCalibrationPoints( List<double> rawValues,
                           List<double> refValues)

java int loadCalibrationPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)

py   def loadCalibrationPoints( rawValues, refValues)
cmd  YCompass target loadCalibrationPoints rawValues refValues

```

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

compass→load_async()

YCompass

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

```
js function load_async( msValidity, callback, context)
nodejs 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.

compass→nextCompass()**YCompass**

Continues the enumeration of compasses started using `yFirstCompass()`.

js	<code>function nextCompass()</code>
node.js	<code>function nextCompass()</code>
php	<code>function nextCompass()</code>
cpp	<code>YCompass * nextCompass()</code>
m	<code>-(YCompass*) nextCompass</code>
pas	<code>function nextCompass(): TYCompass</code>
vb	<code>function nextCompass() As YCompass</code>
cs	<code>YCompass nextCompass()</code>
java	<code>YCompass nextCompass()</code>
py	<code>def nextCompass()</code>

Returns :

a pointer to a `YCompass` object, corresponding to a compass currently online, or a `null` pointer if there are no more compasses to enumerate.

compass→registerTimedReportCallback()**YCompass**

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

js	function registerTimedReportCallback(callback)
nodejs	function registerTimedReportCallback(callback)
php	function registerTimedReportCallback(\$callback)
cpp	int registerTimedReportCallback(YCompassTimedReportCallback callback)
m	- (int) registerTimedReportCallback : (YCompassTimedReportCallback) callback
pas	function registerTimedReportCallback(callback : TYCompassTimedReportCallback): LongInt
vb	function registerTimedReportCallback() As Integer
cs	int registerTimedReportCallback(TimedReportCallback callback)
java	int registerTimedReportCallback(TimedReportCallback callback)
py	def 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.

compass→registerValueCallback()**YCompass**

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

js	<code>function registerValueCallback(callback)</code>
node.js	<code>function registerValueCallback(callback)</code>
php	<code>function registerValueCallback(\$callback)</code>
cpp	<code>int registerValueCallback(YCompassValueCallback callback)</code>
m	<code>-(int) registerValueCallback : (YCompassValueCallback) callback</code>
pas	<code>function registerValueCallback(callback: TYCompassValueCallback): LongInt</code>
vb	<code>function registerValueCallback() As Integer</code>
cs	<code>int registerValueCallback(ValueCallback callback)</code>
java	<code>int registerValueCallback(UpdateCallback callback)</code>
py	<code>def registerValueCallback(callback)</code>

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

Parameters :

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

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

Changes the recorded maximal value observed.

```
js   function set_highestValue( newval)
node.js function set_highestValue( newval)
php  function set_highestValue( $newval)
cpp   int set_highestValue( double newval)
m    -(int) setHighestValue : (double) newval
pas   function set_highestValue( newval: double): integer
vb    function set_highestValue( ByVal newval As Double) As Integer
cs    int set_highestValue( double newval)
java  int set_highestValue( double newval)
py    def set_highestValue( newval)
cmd   YCompass target set_highestValue newval
```

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

compass→set_logFrequency() compass→setLogFrequency()

YCompass

Changes the datalogger recording frequency for this function.

js	<code>function set_logFrequency(newval)</code>
nodejs	<code>function set_logFrequency(newval)</code>
php	<code>function set_logFrequency(\$newval)</code>
cpp	<code>int set_logFrequency(const string& newval)</code>
m	<code>-(int) setLogFrequency : (NSString*) newval</code>
pas	<code>function set_logFrequency(newval: string): integer</code>
vb	<code>function set_logFrequency(ByVal newval As String) As Integer</code>
cs	<code>int set_logFrequency(string newval)</code>
java	<code>int set_logFrequency(String newval)</code>
py	<code>def set_logFrequency(newval)</code>
cmd	<code>YCompass target set_logFrequency newval</code>

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

compass→set_logicalName() compass→setLogicalName()

YCompass

Changes the logical name of the compass.

js	function set_logicalName(newval)
node.js	function set_logicalName(newval)
php	function set_logicalName(\$newval)
cpp	int set_logicalName(const string& newval)
m	- (int) setLogicalName : (NSString*) newval
pas	function set_logicalName(newval: string): integer
vb	function set_logicalName(ByVal newval As String) As Integer
cs	int set_logicalName(string newval)
java	int set_logicalName(String newval)
py	def set_logicalName(newval)
cmd	YCompass target set_logicalName newval

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

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

compass→set_lowestValue() compass→setLowestValue()

YCompass

Changes the recorded minimal value observed.

<code>js</code>	<code>function set_lowestValue(newval)</code>
<code>node.js</code>	<code>function set_lowestValue(newval)</code>
<code>php</code>	<code>function set_lowestValue(\$newval)</code>
<code>cpp</code>	<code>int set_lowestValue(double newval)</code>
<code>m</code>	<code>-(int) setLowestValue : (double) newval</code>
<code>pas</code>	<code>function set_lowestValue(newval: double): integer</code>
<code>vb</code>	<code>function set_lowestValue(ByVal newval As Double) As Integer</code>
<code>cs</code>	<code>int set_lowestValue(double newval)</code>
<code>java</code>	<code>int set_lowestValue(double newval)</code>
<code>py</code>	<code>def set_lowestValue(newval)</code>
<code>cmd</code>	<code>YCompass target set_lowestValue newval</code>

Parameters :

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

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

Changes the timed value notification frequency for this function.

js	function set_reportFrequency(newval)
node.js	function set_reportFrequency(newval)
php	function set_reportFrequency(\$newval)
cpp	int set_reportFrequency(const string& newval)
m	- (int) setReportFrequency : (NSString*) newval
pas	function set_reportFrequency(newval: string): integer
vb	function set_reportFrequency(ByVal newval As String) As Integer
cs	int set_reportFrequency(string newval)
java	int set_reportFrequency(String newval)
py	def set_reportFrequency(newval)
cmd	YCompass target set_reportFrequency newval

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

compass→set_resolution() compass→setResolution()

YCompass

Changes the resolution of the measured physical values.

js	function set_resolution(newval)
nodejs	function set_resolution(newval)
php	function set_resolution(\$newval)
cpp	int set_resolution(double newval)
m	-(int) setResolution : (double) newval
pas	function set_resolution(newval: double): integer
vb	function set_resolution(ByVal newval As Double) As Integer
cs	int set_resolution(double newval)
java	int set_resolution(double newval)
py	def set_resolution(newval)
cmd	YCompass target set_resolution newval

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

compass→set(userData)
compass→setUserData()**YCompass**

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

```
js   function set(userData) {  
node.js function set(userData) {  
php  function setUserData( $data ) {  
cpp   void setUserData( void* data ) {  
m     -(void) setUserData : (void*) data  
pas   procedure set(userData: Tobject);  
vb    procedure setUserData( ByVal data As Object );  
cs    void setUserData( object data );  
java  void setUserData( Object data );  
py    def setUserData( 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

compass→wait_async()**YCompass**

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

```
js  function wait_async( callback, context )
nodejs function 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.

20.8. Gyroscope function interface

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

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

```

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

```

Global functions

yFindGyro(func)

Retrieves a gyroscope for a given identifier.

yFirstGyro()

Starts the enumeration of gyroscopes currently accessible.

YGyro methods

gyro→calibrateFromPoints(rawValues, refValues)

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

gyro→describe()

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

gyro→get_advertisedValue()

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

gyro→get_currentRawValue()

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

gyro→get_currentValue()

Returns the current value of the angular velocity, in degrees per second, as a floating point number.

gyro→get_errorMessage()

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

gyro→get_errorType()

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

gyro→get_friendlyName()

Returns a global identifier of the gyroscope in the format MODULE_NAME.FUNCTION_NAME.

gyro→get_functionDescriptor()

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

gyro→get_functionId()

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

gyro→get_hardwareId()

Returns the unique hardware identifier of the gyroscope in the form SERIAL.FUNCTIONID.

gyro→get_heading()

Returns the estimated heading angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_highestValue()

Returns the maximal value observed for the angular velocity since the device was started.

gyro→get_logFrequency()

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

gyro→get_logicalName()

Returns the logical name of the gyroscope.

gyro→get_lowestValue()

Returns the minimal value observed for the angular velocity since the device was started.

gyro→get_module()

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

gyro→get_module_async(callback, context)

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

gyro→get_pitch()

Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionW()

Returns the w component (real part) of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionX()

Returns the x component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionY()

Returns the y component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_quaternionZ()

Returns the z component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_recordedData(startTime, endTime)

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

gyro→get_reportFrequency()

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

gyro→get_resolution()

Returns the resolution of the measured values.

gyro→get_roll()

Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

gyro→get_unit()

Returns the measuring unit for the angular velocity.

gyro→get_userData()

Returns the value of the userData attribute, as previously stored using method set(userData).

gyro→get_xValue()

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

gyro→get_yValue()

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

gyro→get_zValue()

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

gyro→isOnline()

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

gyro→isOnline_async(callback, context)

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

gyro→load(msValidity)

Preloads the gyroscope cache with a specified validity duration.

gyro→loadCalibrationPoints(rawValues, refValues)

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

gyro→load_async(msValidity, callback, context)

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

gyro→nextGyro()

Continues the enumeration of gyroscopes started using yFirstGyro().

gyro→registerAnglesCallback(callback)

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

gyro→registerQuaternionCallback(callback)

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

gyro→registerTimedReportCallback(callback)

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

gyro→registerValueCallback(callback)

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

gyro→set_highestValue(newval)

Changes the recorded maximal value observed.

gyro→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

gyro→set_logicalName(newval)

Changes the logical name of the gyroscope.

gyro→set_lowestValue(newval)

Changes the recorded minimal value observed.

gyro→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

gyro→set_resolution(newval)

Changes the resolution of the measured physical values.

gyro→set_userData(data)

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

gyro→wait_async(callback, context)

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

YGyro.FindGyro() yFindGyro()

YGyro

Retrieves a gyroscope for a given identifier.

js	function yFindGyro(func)
node.js	function FindGyro(func)
php	function yFindGyro(\$func)
cpp	YGyro* yFindGyro(string func)
m	+(YGyro*) FindGyro : (NSString*) func
pas	function yFindGyro(func: string): TYGyro
vb	function yFindGyro(ByVal func As String) As YGyro
cs	YGyro FindGyro(string func)
java	YGyro FindGyro(String func)
py	def FindGyro(func)

The identifier can be specified using several formats:

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

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

Parameters :

func a string that uniquely characterizes the gyroscope

Returns :

a `YGyro` object allowing you to drive the gyroscope.

YGyro.FirstGyro() yFirstGyro()

YGyro

Starts the enumeration of gyroscopes currently accessible.

```
js function yFirstGyro( )  
node.js function FirstGyro( )  
php function yFirstGyro( )  
cpp YGyro* yFirstGyro( )  
m +(YGyro*) FirstGyro  
pas function yFirstGyro( ): TYGyro  
vb function yFirstGyro( ) As YGyro  
cs YGyro FirstGyro( )  
java YGyro FirstGyro( )  
py def FirstGyro( )
```

Use the method `YGyro.nextGyro()` to iterate on next gyroscopes.

Returns :

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

gyro→calibrateFromPoints()**YGYro**

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)
nodejs function calibrateFromPoints( rawValues, refValues)
php  function calibrateFromPoints( $rawValues, $refValues)
cpp   int calibrateFromPoints( vector<double> rawValues,
                               vector<double> refValues)

m    -(int) calibrateFromPoints : (NSMutableArray*) rawValues
                           : (NSMutableArray*) refValues

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

vb   procedure calibrateFromPoints( )

cs   int calibrateFromPoints( List<double> rawValues,
                           List<double> refValues)

java int calibrateFromPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)

py   def calibrateFromPoints( rawValues, refValues)
cmd  YGYro target calibrateFromPoints rawValues refValues

```

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

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

Parameters :

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

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

Returns :

`YAPI_SUCCESS` if the call succeeds.

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

gyro→describe()**YGyro**

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

js	function describe()
nodejs	function describe()
php	function describe()
cpp	string describe()
m	-(NSString*) describe
pas	function describe(): string
vb	function describe() As String
cs	string describe()
java	String describe()
py	def describe()

More precisely, **TYPE** is the type of the function, **NAME** is the name used for the first access to the function, **SERIAL** is the serial number of the module if the module is connected or "unresolved", and **FUNCTIONID** is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYL01-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the gyroscope (ex: `Relay(MyCustomName.relay1)=RELAYL01-123456.relay1`)

gyro→get_advertisedValue()
gyro→advertisedValue()**YGyro**

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

js	function get_advertisedValue()
nodejs	function get_advertisedValue()
php	function get_advertisedValue()
cpp	string get_advertisedValue()
m	-(NSString*) advertisedValue
pas	function get_advertisedValue() : string
vb	function get_advertisedValue() As String
cs	string get_advertisedValue()
java	String get_advertisedValue()
py	def get_advertisedValue()
cmd	YGyro target get_advertisedValue

Returns :

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

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

gyro→get_currentRawValue()
gyro→currentRawValue()**YGyro**

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

```
js function get_currentRawValue( )  
nodejs function get_currentRawValue( )  
php function get_currentRawValue( )  
cpp double get_currentRawValue( )  
m -(double) currentRawValue  
pas function get_currentRawValue( ): double  
vb function get_currentRawValue( ) As Double  
cs double get_currentRawValue( )  
java double get_currentRawValue( )  
py def get_currentRawValue( )  
cmd YGyro target get_currentRawValue
```

Returns :

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

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

gyro→get_currentValue()**YGyro****gyro→currentValue()**

Returns the current value of the angular velocity, in degrees per second, as a floating point number.

js	function get_currentValue()
nodejs	function get_currentValue()
php	function get_currentValue()
cpp	double get_currentValue()
m	-(double) currentValue
pas	function get_currentValue() : double
vb	function get_currentValue() As Double
cs	double get_currentValue()
java	double get_currentValue()
py	def get_currentValue()
cmd	YGyro target get_currentValue

Returns :

a floating point number corresponding to the current value of the angular velocity, in degrees per second, as a floating point number

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

gyro→get_errorMessage()
gyro→errorMessage()**YGyro**

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

```
js function get_errorMessage( )
node.js function get_errorMessage( )
php function get_errorMessage( )
cpp string get_errorMessage( )
m -(NSString*) errorMessage
pas function get_errorMessage( ): string
vb function get_errorMessage( ) As String
cs string get_errorMessage( )
java String get_errorMessage( )
py def get_errorMessage( )
```

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

Returns :

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

gyro→get_errorType() gyro→errorType()

YGyro

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

js	function get_errorType()
node.js	function get_errorType()
php	function get_errorType()
cpp	YRETCODE get_errorType()
pas	function get_errorType() : YRETCODE
vb	function get_errorType() As YRETCODE
cs	YRETCODE get_errorType()
java	int get_errorType()
py	def get_errorType()

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

Returns :

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

gyro→get_friendlyName()**YGyro**

Returns a global identifier of the gyroscope in the format MODULE_NAME . FUNCTION_NAME.

```
js function get_friendlyName( )  
node.js function get_friendlyName( )  
php function get_friendlyName( )  
cpp string get_friendlyName( )  
m -(NSString*) friendlyName  
cs string get_friendlyName( )  
java String get_friendlyName( )  
py def get_friendlyName( )
```

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

Returns :

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

On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

gyro→get_functionDescriptor() gyro→functionDescriptor()

YGyro

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

js	<code>function get_functionDescriptor()</code>
nodejs	<code>function get_functionDescriptor()</code>
php	<code>function get_functionDescriptor()</code>
cpp	<code>YFUN_DESCR get_functionDescriptor()</code>
m	<code>-(YFUN_DESCR) functionDescriptor</code>
pas	<code>function get_functionDescriptor(): YFUN_DESCR</code>
vb	<code>function get_functionDescriptor() As YFUN_DESCR</code>
cs	<code>YFUN_DESCR get_functionDescriptor()</code>
java	<code>String get_functionDescriptor()</code>
py	<code>def get_functionDescriptor()</code>

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

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

gyro→get_functionId()
gyro→functionId()**YGyro**

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

js	function get_functionId()
node.js	function get_functionId()
php	function get_functionId()
cpp	string get_functionId()
m	-(NSString*) functionId
vb	function get_functionId() As String
cs	string get_functionId()
java	String get_functionId()
py	def get_functionId()

For example `relay1`

Returns :

a string that identifies the gyroscope (ex: `relay1`)

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

gyro→get.hardwareId() gyro→hardwareId()

YGyro

Returns the unique hardware identifier of the gyroscope in the form SERIAL.FUNCTIONID.

js	function get.hardwareId()
nodejs	function get.hardwareId()
php	function get.hardwareId()
cpp	string get.hardwareId()
m	-(NSString*) hardwareId
vb	function get.hardwareId() As String
cs	string get.hardwareId()
java	String get.hardwareId()
py	def get.hardwareId()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the gyroscope (for example RELAYL01-123456.relay1).

Returns :

a string that uniquely identifies the gyroscope (ex: RELAYL01-123456.relay1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

gyro→get_heading()
gyro→heading()**YGyro**

Returns the estimated heading angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
js function get_heading( )
nodejs function get_heading( )
php function get_heading( )
cpp double get_heading( )
m -(double) heading
pas function get_heading( ): double
vb function get_heading( ) As Double
cs double get_heading( )
java double get_heading( )
py def get_heading( )
```

The axis corresponding to the heading can be mapped to any of the device X, Y or Z physical directions using methods of the class `YRefFrame`.

Returns :

a floating-point number corresponding to heading in degrees, between 0 and 360.

gyro→get_highestValue() gyro→highestValue()

YGyro

Returns the maximal value observed for the angular velocity since the device was started.

```
js   function get_highestValue( )  
nodejs function get_highestValue( )  
php  function get_highestValue( )  
cpp   double get_highestValue( )  
m    -(double) highestValue  
pas   function get_highestValue( ): double  
vb    function get_highestValue( ) As Double  
cs    double get_highestValue( )  
java  double get_highestValue( )  
py    def get_highestValue( )  
cmd   YGyro target get_highestValue
```

Returns :

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

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

gyro→get_logFrequency()**YGyro**

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( )  
nodejs function get_logFrequency( )  
php  function get_logFrequency( )  
cpp   string get_logFrequency( )  
m    -(NSString*) logFrequency  
pas   function get_logFrequency( ):string  
vb    function get_logFrequency( ) As String  
cs    string get_logFrequency( )  
java  String get_logFrequency( )  
py    def get_logFrequency( )  
cmd  YGyro target get_logFrequency
```

Returns :

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

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

gyro→get_logicalName() gyro→logicalName()

YGYro

Returns the logical name of the gyroscope.

```
js function get_logicalName( )
nodejs function get_logicalName( )
php function get_logicalName( )
cpp string get_logicalName( )
m -(NSString*) logicalName
pas function get_logicalName( ): string
vb function get_logicalName( ) As String
cs string get_logicalName( )
java String get_logicalName( )
py def get_logicalName( )
cmd YGYro target get_logicalName
```

Returns :

a string corresponding to the logical name of the gyroscope.

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

gyro→get_lowestValue()
gyro→lowestValue()**YGyro**

Returns the minimal value observed for the angular velocity since the device was started.

js	function get_lowestValue()
node.js	function get_lowestValue()
php	function get_lowestValue()
cpp	double get_lowestValue()
m	-(double) lowestValue
pas	function get_lowestValue() : double
vb	function get_lowestValue() As Double
cs	double get_lowestValue()
java	double get_lowestValue()
py	def get_lowestValue()
cmd	YGyro target get_lowestValue

Returns :

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

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

gyro→get_module()**YGyro****gyro→module()**

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

js	function get_module()
nodejs	function get_module()
php	function get_module()
cpp	YModule * get_module()
m	-(YModule*) module
pas	function get_module() : TYModule
vb	function get_module() As YModule
cs	YModule get_module()
java	YModule get_module()
py	def get_module()

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

Returns :

an instance of YModule

gyro→get_module_async()
gyro→module_async()**YGyro**

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

```
js   function get_module_async( callback, context )
nodejs 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 Javascript 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.

gyro→get_pitch()
gyro→pitch()**YGyro**

Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
js function get_pitch( )
nodejs function get_pitch( )
php function get_pitch( )
cpp double get_pitch( )
m -(double) pitch
pas function get_pitch( ): double
vb function get_pitch( ) As Double
cs double get_pitch( )
java double get_pitch( )
py def get_pitch( )
```

The axis corresponding to the pitch angle can be mapped to any of the device X, Y or Z physical directions using methods of the class `YRefFrame`.

Returns :

a floating-point number corresponding to pitch angle in degrees, between -90 and +90.

gyro→get_quaternionW()**YGyro**

Returns the w component (real part) of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
js   function get_quaternionW( )
nodejs function get_quaternionW( )
php  function get_quaternionW( )
cpp   double get_quaternionW( )
m    -(double) quaternionW
pas   function get_quaternionW( ): double
vb    function get_quaternionW( ) As Double
cs   double get_quaternionW( )
java  double get_quaternionW( )
py    def get_quaternionW( )
```

Returns :

a floating-point number corresponding to the w component of the quaternion.

gyro→get_quaternionX() gyro→quaternionX()

YGyro

Returns the `x` component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
js   function get_quaternionX( )
nodejs function get_quaternionX( )
php  function get_quaternionX( )
cpp   double get_quaternionX( )
m    -(double) quaternionX
pas   function get_quaternionX( ): double
vb    function get_quaternionX( ) As Double
cs    double get_quaternionX( )
java  double get_quaternionX( )
py    def get_quaternionX( )
```

The `x` component is mostly correlated with rotations on the roll axis.

Returns :

a floating-point number corresponding to the `x` component of the quaternion.

gyro→get_quaternionY()
gyro→quaternionY()**YGyro**

Returns the y component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

```
js   function get_quaternionY( )
nodejs function get_quaternionY( )
php  function get_quaternionY( )
cpp   double get_quaternionY( )
m    -(double) quaternionY
pas   function get_quaternionY( ): double
vb    function get_quaternionY( ) As Double
cs   double get_quaternionY( )
java  double get_quaternionY( )
py    def get_quaternionY( )
```

The y component is mostly correlated with rotations on the pitch axis.

Returns :

a floating-point number corresponding to the y component of the quaternion.

gyro→get_quaternionZ()

gyro→quaternionZ()

YGyro

Returns the `x` component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

<code>js</code>	<code>function get_quaternionZ()</code>
<code>nodejs</code>	<code>function get_quaternionZ()</code>
<code>php</code>	<code>function get_quaternionZ()</code>
<code>cpp</code>	<code>double get_quaternionZ()</code>
<code>m</code>	<code>-(double) quaternionZ</code>
<code>pas</code>	<code>function get_quaternionZ(): double</code>
<code>vb</code>	<code>function get_quaternionZ() As Double</code>
<code>cs</code>	<code>double get_quaternionZ()</code>
<code>java</code>	<code>double get_quaternionZ()</code>
<code>py</code>	<code>def get_quaternionZ()</code>

The `x` component is mostly correlated with changes of heading.

Returns :

a floating-point number corresponding to the `z` component of the quaternion.

gyro→get_recordedData()

gyro→recordedData()

YGyro

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

js	function get_recordedData(startTime, endTime)
node.js	function get_recordedData(startTime, endTime)
php	function get_recordedData(\$startTime, \$endTime)
cpp	YDataSet get_recordedData(s64 startTime, s64 endTime)
m	- (YDataSet*) recordedData : (s64) startTime : (s64) endTime
pas	function get_recordedData(startTime: int64, endTime: int64) : TYDataSet
vb	function get_recordedData() As YDataSet
cs	YDataSet get_recordedData(long startTime, long endTime)
java	YDataSet get_recordedData(long startTime, long endTime)
py	def get_recordedData(startTime, endTime)
cmd	YGyro target get_recordedData startTime endTime

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

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

Parameters :

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

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

Returns :

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

gyro→get_reportFrequency()**YGyro****gyro→reportFrequency()**

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

js	function get_reportFrequency()
node.js	function get_reportFrequency()
php	function get_reportFrequency()
cpp	string get_reportFrequency()
m	-(NSString*) reportFrequency
pas	function get_reportFrequency() : string
vb	function get_reportFrequency() As String
cs	string get_reportFrequency()
java	String get_reportFrequency()
py	def get_reportFrequency()
cmd	YGyro target get_reportFrequency

Returns :

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

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

gyro→get_resolution()
gyro→resolution()**YGyro**

Returns the resolution of the measured values.

```
js function get_resolution( )
node.js function get_resolution( )
php function get_resolution( )
cpp double get_resolution( )
m -(double) resolution
pas function get_resolution( ): double
vb function get_resolution( ) As Double
cs double get_resolution( )
java double get_resolution( )
py def get_resolution( )
cmd YGyro target get_resolution
```

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

Returns :

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

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

gyro→get_roll()
gyro→roll()**YGyro**

Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

js	function get_roll()
nodejs	function get_roll()
php	function get_roll()
cpp	double get_roll()
m	-(double) roll
pas	function get_roll() : double
vb	function get_roll() As Double
cs	double get_roll()
java	double get_roll()
py	def get_roll()

The axis corresponding to the roll angle can be mapped to any of the device X, Y or Z physical directions using methods of the class **YRefFrame**.

Returns :

a floating-point number corresponding to roll angle in degrees, between -180 and +180.

gyro→get_unit()
gyro→unit()**YGyro**

Returns the measuring unit for the angular velocity.

```
js function get_unit( )
node.js function get_unit( )
php function get_unit( )
cpp string get_unit( )
m -(NSString*) unit
pas function get_unit( ): string
vb function get_unit( ) As String
cs string get_unit( )
java String get_unit( )
py def get_unit( )
cmd YGyro target get_unit
```

Returns :

a string corresponding to the measuring unit for the angular velocity

On failure, throws an exception or returns Y_UNIT_INVALID.

gyro→get(userData) gyro→userData()

YGyro

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

js	<code>function get(userData) </code>
nodejs	<code>function get(userData) </code>
php	<code>function get(userData) </code>
cpp	<code>void * get(userData) </code>
m	<code>-(void*) userData</code>
pas	<code>function get(userData): Tobject</code>
vb	<code>function get(userData) As Object</code>
cs	<code>object get(userData) </code>
java	<code>Object get(userData) </code>
py	<code>def get(userData) </code>

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

gyro→get_xValue()
gyro→xValue()**YGyro**

Returns the angular velocity around the X axis of the device, as a floating point number.

```
js   function get_xValue( )  
node.js function get_xValue( )  
php  function get_xValue( )  
cpp   double get_xValue( )  
m    -(double) xValue  
pas   function get_xValue( ): double  
vb    function get_xValue( ) As Double  
cs   double get_xValue( )  
java  double get_xValue( )  
py    def get_xValue( )
```

Returns :

a floating point number corresponding to the angular velocity around the X axis of the device, as a floating point number

On failure, throws an exception or returns Y_XVALUE_INVALID.

gyro→get_yValue() gyro→yValue()

YGyro

Returns the angular velocity around the Y axis of the device, as a floating point number.

js	function get_yValue()
node.js	function get_yValue()
php	function get_yValue()
cpp	double get_yValue()
m	-(double) yValue
pas	function get_yValue() : double
vb	function get_yValue() As Double
cs	double get_yValue()
java	double get_yValue()
py	def get_yValue()

Returns :

a floating point number corresponding to the angular velocity around the Y axis of the device, as a floating point number

On failure, throws an exception or returns `Y_YVALUE_INVALID`.

gyro→get_zValue()
gyro→zValue()**YGyro**

Returns the angular velocity around the Z axis of the device, as a floating point number.

```
js function get_zValue( )  
node.js function get_zValue( )  
php function get_zValue( )  
cpp double get_zValue( )  
m -(double) zValue  
pas function get_zValue( ): double  
vb function get_zValue( ) As Double  
cs double get_zValue( )  
java double get_zValue( )  
py def get_zValue( )
```

Returns :

a floating point number corresponding to the angular velocity around the Z axis of the device, as a floating point number

On failure, throws an exception or returns Y_ZVALUE_INVALID.

gyro→isOnline()

YGyro

Checks if the gyroscope is currently reachable, without raising any error.

js	function isOnline ()
node.js	function isOnline ()
php	function isOnline ()
cpp	bool isOnline ()
m	-(BOOL) isOnline
pas	function isOnline (): boolean
vb	function isOnline () As Boolean
cs	bool isOnline ()
java	boolean isOnline ()
py	def isOnline ()

If there is a cached value for the gyroscope in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the gyroscope.

Returns :

true if the gyroscope can be reached, and false otherwise

gyro→isOnline_async()**YGyro**

Checks if the gyroscope is currently reachable, without raising any error (asynchronous version).

```
js function isOnline_async( callback, context)
nodejs function isOnline_async( callback, context)
```

If there is a cached value for the gyroscope in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the 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.

gyro→load()**YGyro**

Preloads the gyroscope cache with a specified validity duration.

<code>js</code>	<code>function load(msValidity)</code>
<code>node.js</code>	<code>function load(msValidity)</code>
<code>php</code>	<code>function load(\$msValidity)</code>
<code>cpp</code>	<code>YRETCODE load(int msValidity)</code>
<code>m</code>	<code>-(YRETCODE) load : (int) msValidity</code>
<code>pas</code>	<code>function load(msValidity: integer): YRETCODE</code>
<code>vb</code>	<code>function load(ByVal msValidity As Integer) As YRETCODE</code>
<code>cs</code>	<code>YRETCODE load(int msValidity)</code>
<code>java</code>	<code>int load(long msValidity)</code>
<code>py</code>	<code>def load(msValidity)</code>

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→loadCalibrationPoints()**YGyro**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```

js   function loadCalibrationPoints( rawValues, refValues)
nodejs function loadCalibrationPoints( rawValues, refValues)
php  function loadCalibrationPoints( &$rawValues, &$refValues)
cpp   int loadCalibrationPoints( vector<double>& rawValues,
                                vector<double>& refValues)
m    -(int) loadCalibrationPoints : (NSMutableArray*) rawValues
                           : (NSMutableArray*) refValues
pas  function loadCalibrationPoints( var rawValues: TDoubleArray,
                           var refValues: TDoubleArray): LongInt
vb   procedure loadCalibrationPoints( )
cs   int loadCalibrationPoints( List<double> rawValues,
                           List<double> refValues)
java int loadCalibrationPoints( ArrayList<Double> rawValues,
                           ArrayList<Double> refValues)
py   def loadCalibrationPoints( rawValues, refValues)
cmd  YGyro target loadCalibrationPoints rawValues refValues

```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→load_async()**YGyro**

Preloads the gyroscope cache with a specified validity duration (asynchronous version).

js	function load_async(msValidity, callback, context)
node.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.

gyro→nextGyro()**YGyro**

Continues the enumeration of gyroscopes started using `yFirstGyro()`.

js	function nextGyro()
nodejs	function nextGyro()
php	function nextGyro()
cpp	YGyro * nextGyro()
m	-(YGyro*) nextGyro
pas	function nextGyro() : TYGyro
vb	function nextGyro() As YGyro
cs	YGyro nextGyro()
java	YGyro nextGyro()
py	def nextGyro()

Returns :

a pointer to a `YGyro` object, corresponding to a gyroscope currently online, or a `null` pointer if there are no more gyroscopes to enumerate.

gyro→registerAnglesCallback()**YGYro**

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

```

js   function registerAnglesCallback( callback)
nodejs function registerAnglesCallback( callback)
php  function registerAnglesCallback( $callback)
cpp   int registerAnglesCallback( YAnglesCallback callback)
m    -(int) registerAnglesCallback : (YAnglesCallback) callback
pas   function registerAnglesCallback( callback: TYAnglesCallback): LongInt
vb    function registerAnglesCallback( ) As Integer
cs   int registerAnglesCallback( YAnglesCallback callback)
java  int registerAnglesCallback( YAnglesCallback callback)
py    def registerAnglesCallback( callback)
```

The call frequency is typically around 95Hz during a move. The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to invoke, or a null pointer. The callback function should take four arguments: the YGYro object of the turning device, and the floating point values of the three angles roll, pitch and heading in degrees (as floating-point numbers).

gyro→registerQuaternionCallback()**YGyro**

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

```

js   function registerQuaternionCallback( callback)
nodejs function registerQuaternionCallback( callback)
php  function registerQuaternionCallback( $callback)
cpp   int registerQuaternionCallback( YQuatCallback callback)
m    -(int) registerQuaternionCallback : (YQuatCallback) callback
pas   function registerQuaternionCallback( callback: TYQuatCallback): LongInt
vb    function registerQuaternionCallback( ) As Integer
cs   int registerQuaternionCallback( YQuatCallback callback)
java  int registerQuaternionCallback( YQuatCallback callback)
py    def registerQuaternionCallback( callback)

```

The call frequency is typically around 95Hz during a move. The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to invoke, or a null pointer. The callback function should take five arguments: the YGyro object of the turning device, and the floating point values of the four components w, x, y and z (as floating-point numbers).

gyro→registerTimedReportCallback()**YGyro**

Registers the callback function that is invoked on every periodic timed notification.

js	<code>function registerTimedReportCallback(callback)</code>
node.js	<code>function registerTimedReportCallback(callback)</code>
php	<code>function registerTimedReportCallback(\$callback)</code>
cpp	<code>int registerTimedReportCallback(YGyroTimedReportCallback callback)</code>
m	<code>-(int) registerTimedReportCallback : (YGyroTimedReportCallback) callback</code>
pas	<code>function registerTimedReportCallback(callback: TYGyroTimedReportCallback): LongInt</code>
vb	<code>function registerTimedReportCallback() As Integer</code>
cs	<code>int registerTimedReportCallback(TimedReportCallback callback)</code>
java	<code>int registerTimedReportCallback(TimedReportCallback callback)</code>
py	<code>def registerTimedReportCallback(callback)</code>

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

gyro→registerValueCallback()**YGyro**

Registers the callback function that is invoked on every change of advertised value.

<code>js</code>	<code>function registerValueCallback(callback)</code>
<code>node.js</code>	<code>function registerValueCallback(callback)</code>
<code>php</code>	<code>function registerValueCallback(\$callback)</code>
<code>cpp</code>	<code>int registerValueCallback(YGyroValueCallback callback)</code>
<code>m</code>	<code>- (int) registerValueCallback : (YGyroValueCallback) callback</code>
<code>pas</code>	<code>function registerValueCallback(callback: TYGyroValueCallback): LongInt</code>
<code>vb</code>	<code>function registerValueCallback() As Integer</code>
<code>cs</code>	<code>int registerValueCallback(ValueCallback callback)</code>
<code>java</code>	<code>int registerValueCallback(UpdateCallback callback)</code>
<code>py</code>	<code>def registerValueCallback(callback)</code>

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

gyro→set_highestValue() gyro→setHighestValue()

YGYro

Changes the recorded maximal value observed.

<code>js</code>	<code>function set_highestValue(newval)</code>
<code>node.js</code>	<code>function set_highestValue(newval)</code>
<code>php</code>	<code>function set_highestValue(\$newval)</code>
<code>cpp</code>	<code>int set_highestValue(double newval)</code>
<code>m</code>	<code>-(int) setHighestValue : (double) newval</code>
<code>pas</code>	<code>function set_highestValue(newval: double): integer</code>
<code>vb</code>	<code>function set_highestValue(ByVal newval As Double) As Integer</code>
<code>cs</code>	<code>int set_highestValue(double newval)</code>
<code>java</code>	<code>int set_highestValue(double newval)</code>
<code>py</code>	<code>def set_highestValue(newval)</code>
<code>cmd</code>	<code>YGYro target set_highestValue newval</code>

Parameters :

`newval` a floating point number corresponding to the recorded maximal value observed

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_logFrequency()
gyro→setLogFrequency()**YGyro**

Changes the datalogger recording frequency for this function.

js	function set_logFrequency(newval)
node.js	function set_logFrequency(newval)
php	function set_logFrequency(\$newval)
cpp	int set_logFrequency(const string& newval)
m	- (int) setLogFrequency : (NSString*) newval
pas	function set_logFrequency(newval: string): integer
vb	function set_logFrequency(ByVal newval As String) As Integer
cs	int set_logFrequency(string newval)
java	int set_logFrequency(String newval)
py	def set_logFrequency(newval)
cmd	YGyro target set_logFrequency newval

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF".

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_logicalName() gyro→setLogicalName()

YGYro

Changes the logical name of the gyroscope.

<code>js</code>	<code>function set_logicalName(newval)</code>
<code>node.js</code>	<code>function set_logicalName(newval)</code>
<code>php</code>	<code>function set_logicalName(\$newval)</code>
<code>cpp</code>	<code>int set_logicalName(const string& newval)</code>
<code>m</code>	<code>-(int) setLogicalName : (NSString*) newval</code>
<code>pas</code>	<code>function set_logicalName(newval: string): integer</code>
<code>vb</code>	<code>function set_logicalName(ByVal newval As String) As Integer</code>
<code>cs</code>	<code>int set_logicalName(string newval)</code>
<code>java</code>	<code>int set_logicalName(String newval)</code>
<code>py</code>	<code>def set_logicalName(newval)</code>
<code>cmd</code>	<code>YGYro target set_logicalName newval</code>

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

`newval` a string corresponding to the logical name of the gyroscope.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_lowestValue() gyro→setLowestValue()

YGyro

Changes the recorded minimal value observed.

<code>js</code>	<code>function set_lowestValue(newval)</code>
<code>node.js</code>	<code>function set_lowestValue(newval)</code>
<code>php</code>	<code>function set_lowestValue(\$newval)</code>
<code>cpp</code>	<code>int set_lowestValue(double newval)</code>
<code>m</code>	<code>-(int) setLowestValue : (double) newval</code>
<code>pas</code>	<code>function set_lowestValue(newval: double): integer</code>
<code>vb</code>	<code>function set_lowestValue(ByVal newval As Double) As Integer</code>
<code>cs</code>	<code>int set_lowestValue(double newval)</code>
<code>java</code>	<code>int set_lowestValue(double newval)</code>
<code>py</code>	<code>def set_lowestValue(newval)</code>
<code>cmd</code>	<code>YGyro target set_lowestValue newval</code>

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_reportFrequency() gyro→setReportFrequency()

YGYro

Changes the timed value notification frequency for this function.

js	function set_reportFrequency(newval)
nodejs	function set_reportFrequency(newval)
php	function set_reportFrequency(\$newval)
cpp	int set_reportFrequency(const string& newval)
m	-(int) setReportFrequency : (NSString*) newval
pas	function set_reportFrequency(newval: string): integer
vb	function set_reportFrequency(ByVal newval As String) As Integer
cs	int set_reportFrequency(string newval)
java	int set_reportFrequency(String newval)
py	def set_reportFrequency(newval)
cmd	YGYro target set_reportFrequency newval

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable timed value notifications for this function, use the value "OFF".

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set_resolution()
gyro→setResolution()**YGyro**

Changes the resolution of the measured physical values.

js	function set_resolution(newval)
node.js	function set_resolution(newval)
php	function set_resolution(\$newval)
cpp	int set_resolution(double newval)
m	- (int) setResolution : (double) newval
pas	function set_resolution(newval: double): integer
vb	function set_resolution(ByVal newval As Double) As Integer
cs	int set_resolution(double newval)
java	int set_resolution(double newval)
py	def set_resolution(newval)
cmd	YGyro target set_resolution newval

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

gyro→set(userData)
gyro→setUserData()**YGyro**

Stores a user context provided as argument in the userData attribute of the function.

js	function set(userData)
node.js	function set(userData)
php	function set(userData \$data)
cpp	void set(userData void* data)
m	-(void) setUserData : (void*) data
pas	procedure set(userData Tobject)
vb	procedure set(userData ByVal data As Object)
cs	void set(userData object data)
java	void set(userData Object data)
py	def set(userData data)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

gyro→wait_async()**YGyro**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js  function wait_async( callback, context)
nodejs function 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.

20.9. DataLogger function interface

Yoctopuce sensors include a non-volatile memory capable of storing ongoing measured data automatically, without requiring a permanent connection to a computer. The DataLogger function controls the global parameters of the internal data logger.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_datalogger.js'></script>
node.js	var yoctolib = require('yoctolib');
	var YDataLogger = yoctolib.YDataLogger;
php	require_once('yocto_datalogger.php');
cpp	#include "yocto_datalogger.h"
m	#import "yocto_datalogger.h"
pas	uses yocto_datalogger;
vb	yocto_datalogger.vb
cs	yocto_datalogger.cs
java	import com.yoctopuce.YoctoAPI.YDataLogger;
py	from yocto_datalogger import *

Global functions

yFindDataLogger(func)

Retrieves a data logger for a given identifier.

yFirstDataLogger()

Starts the enumeration of data loggers currently accessible.

YDataLogger methods

datalogger→describe()

Returns a short text that describes unambiguously the instance of the data logger in the form TYPE (NAME)=SERIAL.FUNCTIONID.

datalogger→forgetAllDataStreams()

Clears the data logger memory and discards all recorded data streams.

datalogger→get_advertisedValue()

Returns the current value of the data logger (no more than 6 characters).

datalogger→get_autoStart()

Returns the default activation state of the data logger on power up.

datalogger→get_beaconDriven()

Return true if the data logger is synchronised with the localization beacon.

datalogger→get_currentRunIndex()

Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.

datalogger→get_dataSets()

Returns a list of YDataSet objects that can be used to retrieve all measures stored by the data logger.

datalogger→get_dataStreams(v)

Builds a list of all data streams hold by the data logger (legacy method).

datalogger→get_errorMessage()

Returns the error message of the latest error with the data logger.

datalogger→get_errorType()

Returns the numerical error code of the latest error with the data logger.

datalogger→get_friendlyName()

Returns a global identifier of the data logger in the format MODULE_NAME . FUNCTION_NAME.

datalogger→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

datalogger→get_functionId()

Returns the hardware identifier of the data logger, without reference to the module.

datalogger→get_hardwareId()

Returns the unique hardware identifier of the data logger in the form SERIAL . FUNCTIONID.

datalogger→get_logicalName()

Returns the logical name of the data logger.

datalogger→get_module()

Gets the YModule object for the device on which the function is located.

datalogger→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

datalogger→get_recording()

Returns the current activation state of the data logger.

datalogger→get_timeUTC()

Returns the Unix timestamp for current UTC time, if known.

datalogger→get_userData()

Returns the value of the userData attribute, as previously stored using method set(userData).

datalogger→isOnline()

Checks if the data logger is currently reachable, without raising any error.

datalogger→isOnline_async(callback, context)

Checks if the data logger is currently reachable, without raising any error (asynchronous version).

datalogger→load(msValidity)

Preloads the data logger cache with a specified validity duration.

datalogger→load_async(msValidity, callback, context)

Preloads the data logger cache with a specified validity duration (asynchronous version).

datalogger→nextDataLogger()

Continues the enumeration of data loggers started using yFirstDataLogger().

datalogger→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

datalogger→set_autoStart(newval)

Changes the default activation state of the data logger on power up.

datalogger→set_beaconDriven(newval)

Changes the type of synchronisation of the data logger.

datalogger→set_logicalName(newval)

Changes the logical name of the data logger.

datalogger→set_recording(newval)

Changes the activation state of the data logger to start/stop recording data.

datalogger→set_timeUTC(newval)

Changes the current UTC time reference used for recorded data.

datalogger→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

datalogger→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YDataLogger.FindDataLogger() yFindDataLogger()

YDataLogger

Retrieves a data logger for a given identifier.

js	function yFindDataLogger(func)
node.js	function FindDataLogger(func)
php	function yFindDataLogger(\$func)
cpp	YDataLogger* yFindDataLogger(string func)
m	+ (YDataLogger*) FindDataLogger : (NSString*) func
pas	function yFindDataLogger(func: string): TYDataLogger
vb	function yFindDataLogger(ByVal func As String) As YDataLogger
cs	YDataLogger FindDataLogger(string func)
java	YDataLogger FindDataLogger(String func)
py	def 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.

Parameters :

`func` a string that uniquely characterizes the data logger

Returns :

a `YDataLogger` object allowing you to drive the data logger.

YDataLogger.FirstDataLogger() yFirstDataLogger()

YDataLogger

Starts the enumeration of data loggers currently accessible.

```
js function yFirstDataLogger( )
nodejs function FirstDataLogger( )
php function yFirstDataLogger( )
cpp YDataLogger* yFirstDataLogger( )
m +(YDataLogger*) FirstDataLogger
pas function yFirstDataLogger( ): TYDataLogger
vb function yFirstDataLogger( ) As YDataLogger
cs YDataLogger FirstDataLogger( )
java YDataLogger FirstDataLogger( )
py def 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.

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()
nodejs	function describe()
php	function describe()
cpp	string describe()
m	-(NSString*) describe
pas	function describe(): string
vb	function describe() As String
cs	string describe()
java	String describe()
py	def describe()

More precisely, **TYPE** is the type of the function, **NAME** is the name used for the first access to the function, **SERIAL** is the serial number of the module if the module is connected or "unresolved", and **FUNCTIONID** is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYL01-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the data logger (ex: `Relay(MyCustomName.relay1)=RELAYL01-123456.relay1`)

datalogger→forgetAllDataStreams()**YDataLogger**

Clears the data logger memory and discards all recorded data streams.

<code>js</code>	<code>function forgetAllDataStreams()</code>
<code>node.js</code>	<code>function forgetAllDataStreams()</code>
<code>php</code>	<code>function forgetAllDataStreams()</code>
<code>cpp</code>	<code>int forgetAllDataStreams()</code>
<code>m</code>	<code>-(int) forgetAllDataStreams</code>
<code>pas</code>	<code>function forgetAllDataStreams(): LongInt</code>
<code>vb</code>	<code>function forgetAllDataStreams() As Integer</code>
<code>cs</code>	<code>int forgetAllDataStreams()</code>
<code>java</code>	<code>int forgetAllDataStreams()</code>
<code>py</code>	<code>def forgetAllDataStreams()</code>
<code>cmd</code>	<code>YDataLogger target forgetAllDataStreams</code>

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( )  
node.js function get_advertisedValue( )  
php function get_advertisedValue( )  
cpp string get_advertisedValue( )  
m -(NSString*) advertisedValue  
pas function get_advertisedValue( ): string  
vb function get_advertisedValue( ) As String  
cs string get_advertisedValue( )  
java String get_advertisedValue( )  
py def get_advertisedValue( )  
cmd 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()
node.js	function get_autoStart()
php	function get_autoStart()
cpp	Y_AUTOSTART_enum get_autoStart()
m	-(Y_AUTOSTART_enum) autoStart
pas	function get_autoStart() : Integer
vb	function get_autoStart() As Integer
cs	int get_autoStart()
java	int get_autoStart()
py	def 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**

Return true if the data logger is synchronised with the localization beacon.

js	function get_beaconDriven()
node.js	function get_beaconDriven()
php	function get_beaconDriven()
cpp	Y_BEACONDRIVEN_enum get_beaconDriven()
m	-(Y_BEACONDRIVEN_enum) beaconDriven
pas	function get_beaconDriven(): Integer
vb	function get_beaconDriven() As Integer
cs	int get_beaconDriven()
java	int get_beaconDriven()
py	def get_beaconDriven()
cmd	YDataLogger target get_beaconDriven

Returns :

either Y_BEACONDRIVEN_OFF or Y_BEACONDRIVEN_ON

On failure, throws an exception or returns Y_BEACONDRIVEN_INVALID.

datalogger→get_currentRunIndex() datalogger→currentRunIndex()

YDataLogger

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()
nodejs	function get_currentRunIndex()
php	function get_currentRunIndex()
cpp	int get_currentRunIndex()
m	-(int) currentRunIndex
pas	function get_currentRunIndex() : LongInt
vb	function get_currentRunIndex() As Integer
cs	int get_currentRunIndex()
java	int get_currentRunIndex()
py	def 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()
nodejs	function get_dataSets()
php	function get_dataSets()
cpp	vector<YDataSet> get_dataSets()
m	-NSMutableArray* dataSets
pas	function get_dataSets(): TYDataSetArray
vb	function get_dataSets() As List
cs	List<YDataSet> get_dataSets()
java	ArrayList<YDataSet> get_dataSets()
py	def 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() datalogger→dataStreams()

YDataLogger

Builds a list of all data streams hold by the data logger (legacy method).

js	function get_dataStreams(v)
node.js	function get_dataStreams(v)
php	function get_dataStreams(&\$v)
cpp	int get_dataStreams()
m	-(int) dataStreams : (NSArray**) v
pas	function get_dataStreams(v: Tlist): integer
vb	procedure get_dataStreams(ByVal v As List)
cs	int get_dataStreams(List<YDataStream> v)
java	int get_dataStreams(ArrayList<YDataStream> v)
py	def get_dataStreams(v)

The caller must pass by reference an empty array to hold YDataStream objects, and the function fills it with objects describing available data sequences.

This is the old way to retrieve data from the DataLogger. For new applications, you should rather use `get_dataSets()` method, or call directly `get_recordedData()` on the sensor object.

Parameters :

v an array of YDataStream objects to be filled in

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→get_errorMessage()
datalogger→errorMessage()**YDataLogger**

Returns the error message of the latest error with the data logger.

js	function get_errorMessage()
node.js	function get_errorMessage()
php	function get_errorMessage()
cpp	string get_errorMessage()
m	- (NSString*) errorMessage
pas	function get_errorMessage() : string
vb	function get_errorMessage() As String
cs	string get_errorMessage()
java	String getErrorMessage()
py	def getErrorMessage()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the data logger object

datalogger→get_errorType()
datalogger→errorType()**YDataLogger**

Returns the numerical error code of the latest error with the data logger.

js	function get_errorType()
node.js	function get_errorType()
php	function get_errorType()
cpp	YRETCODE get_errorType()
pas	function get_errorType() : YRETCODE
vb	function get_errorType() As YRETCODE
cs	YRETCODE get_errorType()
java	int get_errorType()
py	def get_errorType()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using 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()
node.js	function get_friendlyName()
php	function get_friendlyName()
cpp	string get_friendlyName()
m	-(NSString*) friendlyName
cs	string get_friendlyName()
java	String get_friendlyName()
py	def 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() datalogger→functionDescriptor()

YDataLogger

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

js	function get_functionDescriptor()
nodejs	function get_functionDescriptor()
php	function get_functionDescriptor()
cpp	YFUN_DESCR get_functionDescriptor()
m	-(YFUN_DESCR) functionDescriptor
pas	function get_functionDescriptor() : YFUN_DESCR
vb	function get_functionDescriptor() As YFUN_DESCR
cs	YFUN_DESCR get_functionDescriptor()
java	String get_functionDescriptor()
py	def get_functionDescriptor()

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

datalogger→get_functionId()
datalogger→functionId()**YDataLogger**

Returns the hardware identifier of the data logger, without reference to the module.

js	function get_functionId()
node.js	function get_functionId()
php	function get_functionId()
cpp	string get_functionId()
m	-(NSString*) functionId
vb	function get_functionId() As String
cs	string get_functionId()
java	String get_functionId()
py	def 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()
datalogger→hardwareId()**YDataLogger**

Returns the unique hardware identifier of the data logger in the form SERIAL.FUNCTIONID.

js	function get.hardwareId()
nodejs	function get.hardwareId()
php	function get.hardwareId()
cpp	string get.hardwareId()
m	-(NSString*) hardwareId
vb	function get.hardwareId() As String
cs	string get.hardwareId()
java	String get.hardwareId()
py	def get.hardwareId()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the data logger (for example RELAYL01-123456.relay1).

Returns :

a string that uniquely identifies the data logger (ex: RELAYL01-123456.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( )
node.js function get_logicalName( )
php function get_logicalName( )
cpp string get_logicalName( )
m -(NSString*) logicalName
pas function get_logicalName( ): string
vb function get_logicalName( ) As String
cs string get_logicalName( )
java String get_logicalName( )
py def get_logicalName( )
cmd 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()
nodejs	function get_module()
php	function get_module()
cpp	YModule * get_module()
m	-(YModule*) module
pas	function get_module() : TYModule
vb	function get_module() As YModule
cs	YModule get_module()
java	YModule get_module()
py	def get_module()

If the function cannot be located on any module, the returned instance of YModule is not shown as online.

Returns :

an instance of YModule

**datalogger→get_module_async()
datalogger→module_async()****YDataLogger**

Gets the `YModule` object for the device on which the function is located (asynchronous version).

```
js   function get_module_async( callback, context )
node.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 Javascript 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()
nodejs	function get_recording()
php	function get_recording()
cpp	Y_RECORDING_enum get_recording()
m	-(Y_RECORDING_enum) recording
pas	function get_recording() : Integer
vb	function get_recording() As Integer
cs	int get_recording()
java	int get_recording()
py	def get_recording()
cmd	YDataLogger target get_recording

Returns :

either Y_RECORDING_OFF or Y_RECORDING_ON, according to the current activation state of the data logger

On failure, throws an exception or returns Y_RECORDING_INVALID.

**datalogger→get_timeUTC()
datalogger→timeUTC()****YDataLogger**

Returns the Unix timestamp for current UTC time, if known.

```
js function get_timeUTC( )
node.js function get_timeUTC( )
php function get_timeUTC( )
cpp s64 get_timeUTC( )
m -(s64) timeUTC
pas function get_timeUTC( ): int64
vb function get_timeUTC( ) As Long
cs long get_timeUTC( )
java long get_timeUTC( )
py def 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(userData)
datalogger→userData()**YDataLogger**

Returns the value of the userData attribute, as previously stored using method `set(userData)`.

js	<code>function get(userData) </code>
nodejs	<code>function get(userData) </code>
php	<code>function get(userData) </code>
cpp	<code>void * get(userData) </code>
m	<code>-(void*) userData</code>
pas	<code>function get(userData): Tobject</code>
vb	<code>function get(userData) As Object</code>
cs	<code>object get(userData) </code>
java	<code>Object get(userData) </code>
py	<code>def get(userData) </code>

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

datalogger→isOnline()**YDataLogger**

Checks if the data logger is currently reachable, without raising any error.

js	function isOnline()
nodejs	function isOnline()
php	function isOnline()
cpp	bool isOnline()
m	- (BOOL) isOnline
pas	function isOnline() : boolean
vb	function isOnline() As Boolean
cs	bool isOnline()
java	boolean isOnline()
py	def isOnline()

If there is a cached value for the 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)
node.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→load()**YDataLogger**

Preloads the data logger cache with a specified validity duration.

<code>js</code>	<code>function load(msValidity)</code>
<code>nodejs</code>	<code>function load(msValidity)</code>
<code>php</code>	<code>function load(\$msValidity)</code>
<code>cpp</code>	<code>YRETCODE load(int msValidity)</code>
<code>m</code>	<code>-(YRETCODE) load : (int) msValidity</code>
<code>pas</code>	<code>function load(msValidity: integer): YRETCODE</code>
<code>vb</code>	<code>function load(ByVal msValidity As Integer) As YRETCODE</code>
<code>cs</code>	<code>YRETCODE load(int msValidity)</code>
<code>java</code>	<code>int load(long msValidity)</code>
<code>py</code>	<code>def load(msValidity)</code>

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→load_async()**YDataLogger**

Preloads the data logger cache with a specified validity duration (asynchronous version).

js	function load_async(msValidity, callback, context)
node.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→nextDataLogger()**YDataLogger**

Continues the enumeration of data loggers started using `yFirstDataLogger()`.

js	<code>function nextDataLogger()</code>
nodejs	<code>function nextDataLogger()</code>
php	<code>function nextDataLogger()</code>
cpp	<code>YDataLogger * nextDataLogger()</code>
m	<code>-(YDataLogger*) nextDataLogger</code>
pas	<code>function nextDataLogger(): TYDataLogger</code>
vb	<code>function nextDataLogger() As YDataLogger</code>
cs	<code>YDataLogger nextDataLogger()</code>
java	<code>YDataLogger nextDataLogger()</code>
py	<code>def nextDataLogger()</code>

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	<code>function registerValueCallback(callback)</code>
node.js	<code>function registerValueCallback(callback)</code>
php	<code>function registerValueCallback(\$callback)</code>
cpp	<code>int registerValueCallback(YDataLoggerValueCallback callback)</code>
m	<code>-(int) registerValueCallback : (YDataLoggerValueCallback) callback</code>
pas	<code>function registerValueCallback(callback: TYDataLoggerValueCallback): LongInt</code>
vb	<code>function registerValueCallback() As Integer</code>
cs	<code>int registerValueCallback(ValueCallback callback)</code>
java	<code>int registerValueCallback(UpdateCallback callback)</code>
py	<code>def registerValueCallback(callback)</code>

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)
node.js	function set_autoStart(newval)
php	function set_autoStart(\$newval)
cpp	int set_autoStart(Y_AUTOSTART_enum newval)
m	-(int) setAutoStart : (Y_AUTOSTART_enum) newval
pas	function set_autoStart(newval: Integer): integer
vb	function set_autoStart(ByVal newval As Integer) As Integer
cs	int set_autoStart(int newval)
java	int set_autoStart(int newval)
py	def set_autoStart(newval)
cmd	YDataLogger target set_autoStart newval

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval either `Y_AUTOSTART_OFF` or `Y_AUTOSTART_ON`, according to the default activation state of the data logger on power up

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→set_beaconDriven() datalogger→setBeaconDriven()

YDataLogger

Changes the type of synchronisation of the data logger.

<code>js</code>	<code>function set_beaconDriven(newval)</code>
<code>node.js</code>	<code>function set_beaconDriven(newval)</code>
<code>php</code>	<code>function set_beaconDriven(\$newval)</code>
<code>cpp</code>	<code>int set_beaconDriven(Y_BEACONDRIVEN_enum newval)</code>
<code>m</code>	<code>-(int) setBeaconDriven : (Y_BEACONDRIVEN_enum) newval</code>
<code>pas</code>	<code>function set_beaconDriven(newval: Integer): integer</code>
<code>vb</code>	<code>function set_beaconDriven(ByVal newval As Integer) As Integer</code>
<code>cs</code>	<code>int set_beaconDriven(int newval)</code>
<code>java</code>	<code>int set_beaconDriven(int newval)</code>
<code>py</code>	<code>def set_beaconDriven(newval)</code>
<code>cmd</code>	<code>YDataLogger target set_beaconDriven newval</code>

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)
node.js	function set_logicalName(newval)
php	function set_logicalName(\$newval)
cpp	int set_logicalName(const string& newval)
m	- (int) setLogicalName : (NSString*) newval
pas	function set_logicalName(newval: string): integer
vb	function set_logicalName(ByVal newval As String) As Integer
cs	int set_logicalName(string newval)
java	int set_logicalName(String newval)
py	def set_logicalName(newval)
cmd	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.

<code>js</code>	<code>function set_recording(newval)</code>
<code>node.js</code>	<code>function set_recording(newval)</code>
<code>php</code>	<code>function set_recording(\$newval)</code>
<code>cpp</code>	<code>int set_recording(Y_RECORDING_enum newval)</code>
<code>m</code>	<code>-(int) setRecording : (Y_RECORDING_enum) newval</code>
<code>pas</code>	<code>function set_recording(newval: Integer): integer</code>
<code>vb</code>	<code>function set_recording(ByVal newval As Integer) As Integer</code>
<code>cs</code>	<code>int set_recording(int newval)</code>
<code>java</code>	<code>int set_recording(int newval)</code>
<code>py</code>	<code>def set_recording(newval)</code>
<code>cmd</code>	<code>YDataLogger target set_recording newval</code>

Parameters :

newval either Y_RECORDING_OFF or Y_RECORDING_ON, according to the activation state of the data logger to start/stop recording data

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

**datalogger→set_timeUTC()
datalogger→setTimeUTC()****YDataLogger**

Changes the current UTC time reference used for recorded data.

js	function set_timeUTC(newval)
node.js	function set_timeUTC(newval)
php	function set_timeUTC(\$newval)
cpp	int set_timeUTC(s64 newval)
m	- (int) setTimeUTC : (s64) newval
pas	function set_timeUTC(newval: int64): integer
vb	function set_timeUTC(ByVal newval As Long) As Integer
cs	int set_timeUTC(long newval)
java	int set_timeUTC(long newval)
py	def 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)
node.js	function set(userData)
php	function set(userData) \$data
cpp	void set(userData) void* data
m	-(void) setUserData : (void*) data
pas	procedure set(userData) Tobject
vb	procedure set(userData) ByVal data As Object
cs	void set(userData) object data
java	void set(userData) Object data
py	def set(userData) data

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

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)
nodejs function 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.

20.10. Recorded data sequence

YDataSet objects make it possible to retrieve a set of recorded measures for a given sensor and a specified time interval. They can be used to load data points with a progress report. When the YDataSet object is instantiated by the `get_recordedData()` function, no data is yet loaded from the module. It is only when the `loadMore()` method is called over and over than data will be effectively loaded from the dataLogger.

A preview of available measures is available using the function `get_preview()` as soon as `loadMore()` has been called once. Measures themselves are available using function `get_measures()` when loaded by subsequent calls to `loadMore()`.

This class can only be used on devices that use a recent firmware, as YDataSet objects are not supported by firmwares older than version 13000.

In order to use the functions described here, you should include:

```

js <script type='text/javascript' src='yocto_api.js'></script>
nodejs var yoctolib = require('yoctolib');
var YAPI = yoctolib.YAPI;
var YModule = yoctolib.YModule;
php require_once('yocto_api.php');
cpp #include "yocto_api.h"
m #import "yocto_api.h"
pas uses yocto_api;
vb yocto_api.vb
cs yocto_api.cs
java import com.yoctopuce.YoctoAPI.YModule;
py from yocto_api import *

```

YDataSet methods

`dataset→get_endTimeUTC()`

Returns the end time of the dataset, relative to the Jan 1, 1970.

`dataset→get_functionId()`

Returns the hardware identifier of the function that performed the measure, without reference to the module.

`dataset→get_hardwareId()`

Returns the unique hardware identifier of the function who performed the measures, in the form SERIAL.FUNCTIONID.

`dataset→get_measures()`

Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

`dataset→get_preview()`

Returns a condensed version of the measures that can retrieved in this YDataSet, as a list of YMeasure objects.

`dataset→get_progress()`

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

`dataset→get_startTimeUTC()`

Returns the start time of the dataset, relative to the Jan 1, 1970.

`dataset→get_summary()`

Returns an YMeasure object which summarizes the whole DataSet.

`dataset→get_unit()`

Returns the measuring unit for the measured value.

dataset→loadMore()

Loads the the next block of measures from the dataLogger, and updates the progress indicator.

dataset→loadMore_async(callback, context)

Loads the the next block of measures from the dataLogger asynchronously.

dataset→get_endTimeUTC()**dataset→endTimeUTC()****YDataSet**

Returns the end time of the dataset, relative to the Jan 1, 1970.

js	<code>function get_endTimeUTC()</code>
node.js	<code>function get_endTimeUTC()</code>
php	<code>function get_endTimeUTC()</code>
cpp	<code>s64 get_endTimeUTC()</code>
m	<code>-(s64) endTimeUTC</code>
pas	<code>function get_endTimeUTC(): int64</code>
vb	<code>function get_endTimeUTC() As Long</code>
cs	<code>long get_endTimeUTC()</code>
java	<code>long get_endTimeUTC()</code>
py	<code>def get_endTimeUTC()</code>

When the YDataSet is created, the end time is the value passed in parameter to the `get_dataSet()` function. After the very first call to `loadMore()`, the end time is updated to reflect the timestamp of the last measure actually found in the dataLogger within the specified range.

Returns :

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the end of this data set (i.e. Unix time representation of the absolute time).

**dataset→get_functionId()
dataset→functionId()****YDataSet**

Returns the hardware identifier of the function that performed the measure, without reference to the module.

js	function get_functionId()
nodejs	function get_functionId()
php	function get_functionId()
cpp	string get_functionId()
m	- (NSString*) functionId
pas	function get_functionId() : string
vb	function get_functionId() As String
cs	string get_functionId()
java	String get_functionId()
py	def 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()
node.js	function get_hardwareId()
php	function get_hardwareId()
cpp	string get_hardwareId()
m	-(NSString*) hardwareId
pas	function get_hardwareId() : string
vb	function get_hardwareId() As String
cs	string get_hardwareId()
java	String get_hardwareId()
py	def get_hardwareId()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example THRMCP1-123456.temperature1)

Returns :

a string that uniquely identifies the function (ex: THRMCP1-123456.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( )
node.js function get_measures( )
php  function get_measures( )
cpp   vector<YMeasure> get_measures( )
m    -(NSMutableArray*) measures
pas   function get_measures( ): TYMeasureArray
vb    function get_measures( ) As List
cs    List<YMeasure> get_measures( )
java  ArrayList<YMeasure> get_measures( )
py    def get_measures( )

```

Each item includes: - the start of the measure time interval - the end of the measure time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

Before calling this method, you should call `loadMore()` to load data from the device. You may have to call `loadMore()` several time until all rows are loaded, but you can start looking at available data rows before the load is complete.

The oldest measures are always loaded first, and the most recent measures will be loaded last. As a result, timestamps are normally sorted in ascending order within the measure table, unless there was an unexpected adjustment of the datalogger UTC clock.

Returns :

a table of records, where each record depicts the measured value for a given time interval

On failure, throws an exception or returns an empty array.

dataset→get_preview() 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()
node.js	function get_preview()
php	function get_preview()
cpp	vector<YMeasure> get_preview()
m	-(NSMutableArray*) preview
pas	function get_preview() : TYMeasureArray
vb	function get_preview() As List
cs	List<YMeasure> get_preview()
java	ArrayList<YMeasure> get_preview()
py	def get_preview()

Each item includes: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This preview is available as soon as `loadMore()` has been called for the first time.

Returns :

a table of records, where each record depicts the measured values during a time interval

On failure, throws an exception or returns an empty array.

**dataset→get_progress()
dataset→progress()****YDataSet**

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

js	function get_progress()
nodejs	function get_progress()
php	function get_progress()
cpp	int get_progress()
m	- (int) progress
pas	function get_progress() : LongInt
vb	function get_progress() As Integer
cs	int get_progress()
java	int get_progress()
py	def 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→getStartTimeUTC()**YDataSet****dataset→startTimeUTC()**

Returns the start time of the dataset, relative to the Jan 1, 1970.

js	function getStartTimeUTC()
nodejs	function getStartTimeUTC()
php	function getStartTimeUTC()
cpp	s64 getStartTimeUTC()
m	-(s64) startTimeUTC
pas	function getStartTimeUTC(): int64
vb	function getStartTimeUTC() As Long
cs	long getStartTimeUTC()
java	long getStartTimeUTC()
py	def getStartTimeUTC()

When the YDataSet is created, the start time is the value passed in parameter to the `get_dataSet()` function. After the very first call to `loadMore()`, the start time is updated to reflect the timestamp of the first measure actually found in the dataLogger within the specified range.

Returns :

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the beginning of this data set (i.e. Unix time representation of the absolute time).

**dataset→get_summary()
dataset→summary()****YDataSet**

Returns an YMeasure object which summarizes the whole DataSet.

```
js function get_summary( )
node.js function get_summary( )
php function get_summary( )
cpp YMeasure get_summary( )
m -(YMeasure*) summary
pas function get_summary( ): TYMeasure
vb function get_summary( ) As YMeasure
cs YMeasure get_summary( )
java YMeasure get_summary( )
py def 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_unit()**YDataSet****dataset→unit()**

Returns the measuring unit for the measured value.

js	function get_unit()
node.js	function get_unit()
php	function get_unit()
cpp	string get_unit()
m	-(NSString*) unit
pas	function get_unit() : string
vb	function get_unit() As String
cs	string get_unit()
java	String get_unit()
py	def get_unit()

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( )
nodejs function loadMore( )
php function loadMore( )
cpp int loadMore( )
m -(int) loadMore
pas function loadMore( ): LongInt
vb function loadMore( ) As Integer
cs int loadMore( )
java int loadMore( )
py def 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)
node.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.

20.11. Measured value

YMeasure objects are used within the API to represent a value measured at a specified time. These objects are used in particular in conjunction with the YDataSet class.

In order to use the functions described here, you should include:

```

js <script type='text/javascript' src='yocto_api.js'></script>
nodejs var yoctolib = require('yoctolib');
var YAPI = yoctolib.YAPI;
var YModule = yoctolib.YModule;
php require_once('yocto_api.php');
cpp #include "yocto_api.h"
m #import "yocto_api.h"
pas uses yocto_api;
vb yocto_api.vb
cs yocto_api.cs
java import com.yoctopuce.YoctoAPI.YModule;
py from yocto_api import *

```

YMeasure methods

measure→get_averageValue()

Returns the average value observed during the time interval covered by this measure.

measure→get_endTimeUTC()

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

measure→get_maxValue()

Returns the largest value observed during the time interval covered by this measure.

measure→get_minValue()

Returns the smallest value observed during the time interval covered by this measure.

measure→get_startTimeUTC()

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

measure→get_averageValue()
measure→averageValue()**YMeasure**

Returns the average value observed during the time interval covered by this measure.

js	function get_averageValue()
node.js	function get_averageValue()
php	function get_averageValue()
cpp	double get_averageValue()
m	-(double) averageValue
pas	function get_averageValue(): double
vb	function get_averageValue() As Double
cs	double get_averageValue()
java	double get_averageValue()
py	def 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( )
node.js function get_endTimeUTC( )
php function get_endTimeUTC( )
cpp double get_endTimeUTC( )
m -(double) endTimeUTC
pas function get_endTimeUTC( ): double
vb function get_endTimeUTC( ) As Double
cs double get_endTimeUTC( )
java double get_endTimeUTC( )
py def get_endTimeUTC( )
```

When the recording rate is higher than 1 sample per second, the timestamp may have a fractional part.

Returns :

an floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the end of this measure.

measure→get_maxValue()**YMeasure****measure→maxValue()**

Returns the largest value observed during the time interval covered by this measure.

js	function get_maxValue()
node.js	function get_maxValue()
php	function get_maxValue()
cpp	double get_maxValue()
m	-(double) maxValue
pas	function get_maxValue() : double
vb	function get_maxValue() As Double
cs	double get_maxValue()
java	double get_maxValue()
py	def 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( )
node.js function get_minValue( )
php  function get_minValue( )
cpp   double get_minValue( )
m    -(double) minValue
pas   function get_minValue( ): double
vb    function get_minValue( ) As Double
cs   double get_minValue( )
java  double get_minValue( )
py    def get_minValue( )
```

Returns :

a floating-point number corresponding to the smallest value observed.

measure→getStartTimeUTC()

measure→startTimeUTC()

YMeasure

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

<code>js</code>	<code>function getStartTimeUTC()</code>
<code>node.js</code>	<code>function getStartTimeUTC()</code>
<code>php</code>	<code>function getStartTimeUTC()</code>
<code>cpp</code>	<code>double getStartTimeUTC()</code>
<code>m</code>	<code>-(double) startTimeUTC</code>
<code>pas</code>	<code>function getStartTimeUTC(): double</code>
<code>vb</code>	<code>function getStartTimeUTC() As Double</code>
<code>cs</code>	<code>double getStartTimeUTC()</code>
<code>java</code>	<code>double getStartTimeUTC()</code>
<code>py</code>	<code>def getStartTimeUTC()</code>

When the recording rate is higher than 1 sample per second, the timestamp may have a fractional part.

Returns :

an floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

21. Troubleshooting

21.1. Linux and USB

To work correctly under Linux, 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 Linux¹ archive, there are two rule examples which you can use as a basis.

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

¹ <http://www.yoctopuce.com/FR/virtualhub.php>

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"
```

21.2. ARM Platforms: HF and EL

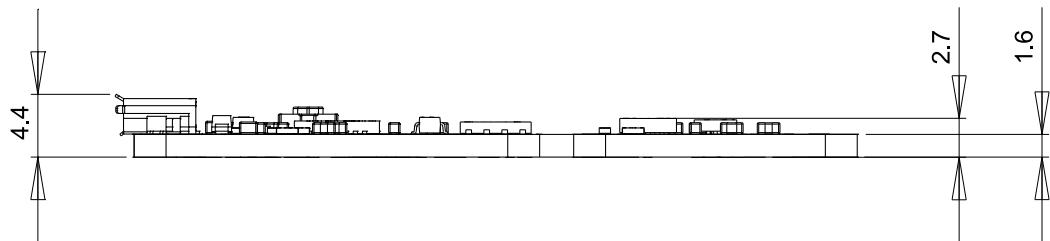
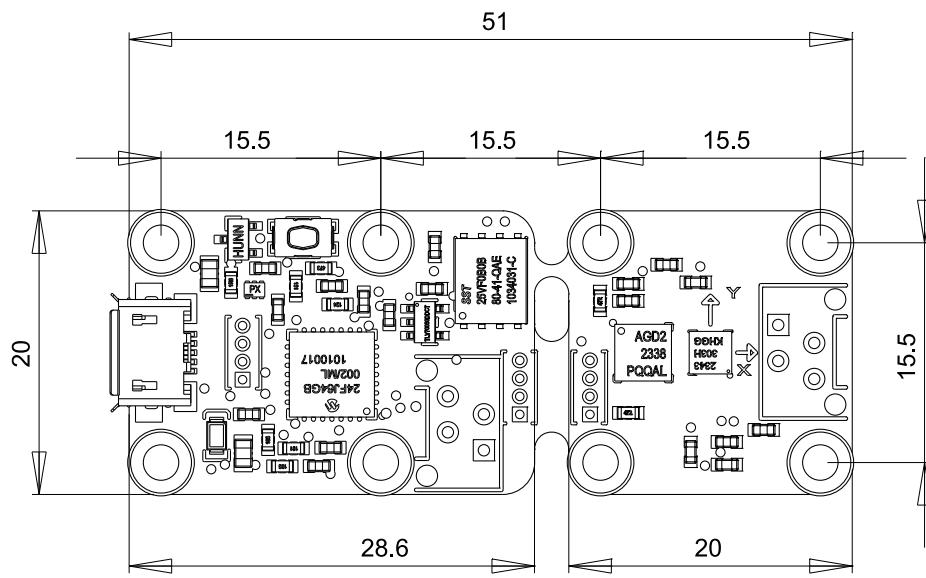
There are two main flavors of executable on ARM: HF (Hard Float) binaries, and EL (EABI Little Endian) binaries. These two families are not compatible at all. The compatibility of a given ARM platform with one of these two families depends on the hardware and on the OS build. ArmHL and ArmEL compatibility 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.

22. Characteristics

You can find below a summary of the main technical characteristics of your Yocto-3D module.

Sensor	LSM303D + L3GD20
Width	20 mm
Length	51 mm
Weight	4 g
USB connector	micro-B
Gyroscopic attitude estimation	95 Hz
Static measures	7 Hz
Measure range (Accel.)	16 g
Measure range (Gyro)	2000 °/s
Measure range (Magn.)	2 gauss
Sensitivity	0.1 °
Sensitivity (Accel.)	0.001 g
Sensitivity (Gyro)	0.1 °/s
Sensitivity (Magn.)	0.001 gauss
Supported Operating Systems	Windows, Linux (Intel + ARM), Mac OS X, Android
Drivers	no driver needed
API / SDK / Libraries (USB+TCP)	C++, Objective-C, C#, VB .NET, Delphi, Python, Java/Android
API / SDK / Libraries (TCP only)	Javascript, Node.js, PHP, Java
RoHS	yes
USB Vendor ID	0x24E0
USB Device ID	0x003B
Suggested enclosure	YoctoBox-3D-Transp



All dimensions are in mm
Toutes les dimensions sont en mm

Yocto-3D

A4

Scale
2:1
Echelle

Index

A

Accelerometer 7, 24, 209
Access 105
Accessories 3
Activating 106
Advanced 123
Android 105, 106
Assembly 15
Attitude 9
Away 16

B

Basic 71
Blueprint 545

C

C# 79
C++ 57, 63
calibrateFromPoints, YAccelerometer 213
calibrateFromPoints, YCompass 388
calibrateFromPoints, YGyro 432
calibrateFromPoints, YMagnetometer 259
calibrateFromPoints, YTilt 345
Calibration 128
Callback 52
cancel3DCalibration, YRefFrame 305
Characteristics 543
checkFirmware, YModule 163
CheckLogicalName, YAPI 135
Command 35, 117
Compass 9, 29, 384
Compatibility 105
Concepts 19
Configuration 12, 27, 301
Connections 15

D

Data 126, 521
DataLogger 30, 483
Delphi 87
describe, YAccelerometer 214
describe, YCompass 389
describe, YDataLogger 487
describe, YGyro 433
describe, YMagnetometer 260
describe, YModule 164
describe, YRefFrame 306
describe, YTilt 346
Description 35
DisableExceptions, YAPI 136
Distribution 17
download, YModule 165

Dynamic 93, 119

E

Elements 5, 6
EnableExceptions, YAPI 137
EnableUSBHost, YAPI 138
Error 44, 55, 62, 69, 76, 84, 92, 97, 104, 115
Estimated 9
Event 123

F

Files 93
Filters 52
FindAccelerometer, YAccelerometer 211
FindCompass, YCompass 386
FindDataLogger, YDataLogger 485
FindGyro, YGyro 430
FindMagnetometer, YMagnetometer 257
FindModule, YModule 161
FindRefFrame, YRefFrame 303
FindTilt, YTilt 343
FirstAccelerometer, YAccelerometer 212
FirstCompass, YCompass 387
FirstDataLogger, YDataLogger 486
FirstGyro, YGyro 431
FirstMagnetometer, YMagnetometer 258
FirstModule, YModule 162
FirstRefFrame, YRefFrame 304
FirstTilt, YTilt 344
Fixing 15
forgetAllDataStreams, YDataLogger 488
Frame 7, 27, 301
FreeAPI, YAPI 139
functionCount, YModule 166
functionId, YModule 167
functionName, YModule 168
Functions 7, 134
functionValue, YModule 169

G

General 19, 35, 134
get_3DCalibrationHint, YRefFrame 307
get_3DCalibrationLogMsg, YRefFrame 308
get_3DCalibrationProgress, YRefFrame 309
get_3DCalibrationStage, YRefFrame 310
get_3DCalibrationStageProgress, YRefFrame 311
get_advertisedValue, YAccelerometer 215
get_advertisedValue, YCompass 390
get_advertisedValue, YDataLogger 489
get_advertisedValue, YGyro 434
get_advertisedValue, YMagnetometer 261
get_advertisedValue, YRefFrame 312

get_advertisedValue, YTilt 347
get_allSettings, YModule 170
get_autoStart, YDataLogger 490
get_averageValue, YMeasure 534
get_beacon, YModule 171
get_beaconDriven, YDataLogger 491
get_bearing, YRefFrame 313
get_currentRawValue, YAccelerometer 216
get_currentRawValue, YCompass 391
get_currentRawValue, YGyro 435
get_currentRawValue, YMagnetometer 262
get_currentRawValue, YTilt 348
get_currentRunIndex, YDataLogger 492
get_currentValue, YAccelerometer 217
get_currentValue, YCompass 392
get_currentValue, YGyro 436
get_currentValue, YMagnetometer 263
get_currentValue, YTilt 349
get_dataSets, YDataLogger 493
get_dataStreams, YDataLogger 494
get_endTimeUTC, YDataSet 522
get_endTimeUTC, YMeasure 535
get_errorMessage, YAccelerometer 218
get_errorMessage, YCompass 393
get_errorMessage, YDataLogger 495
get_errorMessage, YGyro 437
get_errorMessage, YMagnetometer 264
get_errorMessage, YModule 172
get_errorMessage, YRefFrame 314
get_errorMessage, YTilt 350
get_errorType, YAccelerometer 219
get_errorType, YCompass 394
get_errorType, YDataLogger 496
get_errorType, YGyro 438
get_errorType, YMagnetometer 265
get_errorType, YModule 173
get_errorType, YRefFrame 315
get_errorType, YTilt 351
get_firmwareRelease, YModule 174
get_friendlyName, YAccelerometer 220
get_friendlyName, YCompass 395
get_friendlyName, YDataLogger 497
get_friendlyName, YGyro 439
get_friendlyName, YMagnetometer 266
get_friendlyName, YRefFrame 316
get_friendlyName, YTilt 352
get_functionDescriptor, YAccelerometer 221
get_functionDescriptor, YCompass 396
get_functionDescriptor, YDataLogger 498
get_functionDescriptor, YGyro 440
get_functionDescriptor, YMagnetometer 267
get_functionDescriptor, YRefFrame 317
get_functionDescriptor, YTilt 353
get_functionId, YAccelerometer 222
get_functionId, YCompass 397
get_functionId, YDataLogger 499
get_functionId, YDataSet 523
get_functionId, YGyro 441
get_functionId, YMagnetometer 268
get_functionId, YRefFrame 318
get_functionId, YTilt 354
get_hardwareId, YAccelerometer 223
get_hardwareId, YCompass 398
get_hardwareId, YDataLogger 500
get_hardwareId, YDataSet 524
get_hardwareId, YGyro 442
get_hardwareId, YMagnetometer 269
get_hardwareId, YModule 175
get_hardwareId, YRefFrame 319
get_hardwareId, YTilt 355
get_heading, YGyro 443
get_highestValue, YAccelerometer 224
get_highestValue, YCompass 399
get_highestValue, YGyro 444
get_highestValue, YMagnetometer 270
get_highestValue, YTilt 356
get_icon2d, YModule 176
get_lastLogs, YModule 177
get_logFrequency, YAccelerometer 225
get_logFrequency, YCompass 400
get_logFrequency, YGyro 445
get_logFrequency, YMagnetometer 271
get_logFrequency, YTilt 357
get_logicalName, YAccelerometer 226
get_logicalName, YCompass 401
get_logicalName, YDataLogger 501
get_logicalName, YGyro 446
get_logicalName, YMagnetometer 272
get_logicalName, YModule 178
get_logicalName, YRefFrame 320
get_logicalName, YTilt 358
get_lowestValue, YAccelerometer 227
get_lowestValue, YCompass 402
get_lowestValue, YGyro 447
get_lowestValue, YMagnetometer 273
get_lowestValue, YTilt 359
get_luminosity, YModule 179
get_magneticHeading, YCompass 403
get_maxValue, YMeasure 536
get_measures, YDataSet 525
get_minValue, YMeasure 537
get_module, YAccelerometer 228
get_module, YCompass 404
get_module, YDataLogger 502
get_module, YGyro 448
get_module, YMagnetometer 274
get_module, YRefFrame 321
get_module, YTilt 360
get_module_async, YAccelerometer 229
get_module_async, YCompass 405
get_module_async, YDataLogger 503
get_module_async, YGyro 449
get_module_async, YMagnetometer 275
get_module_async, YRefFrame 322
get_module_async, YTilt 361
get_mountOrientation, YRefFrame 323
get_mountPosition, YRefFrame 324
get_persistentSettings, YModule 180

get_pitch, YGyro 450
get_preview, YDataSet 526
get_productId, YModule 181
get_productName, YModule 182
get_productRelease, YModule 183
get_progress, YDataSet 527
get_quaternionW, YGyro 451
get_quaternionX, YGyro 452
get_quaternionY, YGyro 453
get_quaternionZ, YGyro 454
get_rebootCountdown, YModule 184
get_recordedData, YAccelerometer 230
get_recordedData, YCompass 406
get_recordedData, YGyro 455
get_recordedData, YMagnetometer 276
get_recordedData, YTilt 362
get_recording, YDataLogger 504
get_reportFrequency, YAccelerometer 231
get_reportFrequency, YCompass 407
get_reportFrequency, YGyro 456
get_reportFrequency, YMagnetometer 277
get_reportFrequency, YTilt 363
get_resolution, YAccelerometer 232
get_resolution, YCompass 408
get_resolution, YGyro 457
get_resolution, YMagnetometer 278
get_resolution, YTilt 364
get_roll, YGyro 458
get_serialNumber, YModule 185
get_startTimeUTC, YDataSet 528
get_startTimeUTC, YMeasure 538
get_summary, YDataSet 529
get_timeUTC, YDataLogger 505
get_unit, YAccelerometer 233
get_unit, YCompass 409
get_unit, YDataSet 530
get_unit, YGyro 459
get_unit, YMagnetometer 279
get_unit, YTilt 365
get_upTime, YModule 186
get_usbCurrent, YModule 187
get_userData, YAccelerometer 234
get_userData, YCompass 410
get_userData, YDataLogger 506
get_userData, YGyro 460
get_userData, YMagnetometer 280
get_userData, YModule 188
get_userData, YRefFrame 325
get_userData, YTilt 366
get_userVar, YModule 189
get_xValue, YAccelerometer 235
get_xValue, YGyro 461
get_xValue, YMagnetometer 281
get_yValue, YAccelerometer 236
get_yValue, YGyro 462
get_yValue, YMagnetometer 282
get_zValue, YAccelerometer 237
get_zValue, YGyro 463
get_zValue, YMagnetometer 283

GetAPIVersion, YAPI 140
GetTickCount, YAPI 141
Gyroscope 7, 26, 428

H

HandleEvents, YAPI 142
High-level 133
HTTP 52, 117

I

InitAPI, YAPI 143
Installation 71, 79
Installing 35
Integration 63
Interface 159, 209, 255, 341, 384, 428, 483
Introduction 1
isOnline, YAccelerometer 238
isOnline, YCompass 411
isOnline, YDataLogger 507
isOnline, YGyro 464
isOnline, YMagnetometer 284
isOnline, YModule 190
isOnline, YRefFrame 326
isOnline, YTilt 367
isOnline_async, YAccelerometer 239
isOnline_async, YCompass 412
isOnline_async, YDataLogger 508
isOnline_async, YGyro 465
isOnline_async, YMagnetometer 285
isOnline_async, YModule 191
isOnline_async, YRefFrame 327
isOnline_async, YTilt 368

J

Java 99
Javascript 39

L

Languages 117
Libraries 119
Library 63, 93, 122
Limitations 37
Linux 541
load, YAccelerometer 240
load, YCompass 413
load, YDataLogger 509
load, YGyro 466
load, YMagnetometer 286
load, YModule 192
load, YRefFrame 328
load, YTilt 369
load_async, YAccelerometer 242
load_async, YCompass 415
load_async, YDataLogger 510
load_async, YGyro 468
load_async, YMagnetometer 288
load_async, YModule 193

load_async, YRefFrame 329
load_async, YTilt 371
loadCalibrationPoints, YAccelerometer 241
loadCalibrationPoints, YCompass 414
loadCalibrationPoints, YGyro 467
loadCalibrationPoints, YMagnetometer 287
loadCalibrationPoints, YTilt 370
loadMore, YDataSet 531
loadMore_async, YDataSet 532
Localization 11
Logger 126

M

Magnetic 9
Magnetometer 7, 25, 255
Measured 534
Module 11, 20, 23, 36, 41, 50, 60, 67, 74, 82, 89, 95, 101, 111, 159
more3DCalibration, YRefFrame 330
Moving 16

N

Native 30, 105
.NET 71
nextAccelerometer, YAccelerometer 243
nextCompass, YCompass 416
nextDataLogger, YDataLogger 511
nextGyro, YGyro 469
nextMagnetometer, YMagnetometer 289
nextModule, YModule 194
nextRefFrame, YRefFrame 331
nextTilt, YTilt 372

O

Objective-C 65
Optional 3

P

Paradigm 19
Platforms 542
Port 106
Porting 122
Power 17
Preparation 87
PreregisterHub, YAPI 144
Prerequisites 2
Presentation 5
Programming 19, 33, 123
Project 71, 79
Python 93

Q

Quaternion 9

R

reboot, YModule 195
Recorded 521
Reference 7, 27, 133, 301
registerAnglesCallback, YGyro 470
RegisterDeviceArrivalCallback, YAPI 145
RegisterDeviceRemovalCallback, YAPI 146
RegisterHub, YAPI 147
RegisterHubDiscoveryCallback, YAPI 148
registerLogCallback, YModule 196
RegisterLogFunction, YAPI 149
registerQuaternionCallback, YGyro 471
registerTimedReportCallback, YAccelerometer 244
registerTimedReportCallback, YCompass 417
registerTimedReportCallback, YGyro 472
registerTimedReportCallback, YMagnetometer 290
registerTimedReportCallback, YTilt 373
registerValueCallback, YAccelerometer 245
registerValueCallback, YCompass 418
registerValueCallback, YDataLogger 512
registerValueCallback, YGyro 473
registerValueCallback, YMagnetometer 291
registerValueCallback, YRefFrame 332
registerValueCallback, YTilt 374
revertFromFlash, YModule 197

S

save3DCalibration, YRefFrame 333
saveToFlash, YModule 198
SelectArchitecture, YAPI 150
Sensor 16, 128
Sequence 521
Service 30
set_allSettings, YModule 199
set_autoStart, YDataLogger 513
set_beacon, YModule 200
set_beaconDriven, YDataLogger 514
set_bearing, YRefFrame 334
set_highestValue, YAccelerometer 246
set_highestValue, YCompass 419
set_highestValue, YGyro 474
set_highestValue, YMagnetometer 292
set_highestValue, YTilt 375
set_logFrequency, YAccelerometer 247
set_logFrequency, YCompass 420
set_logFrequency, YGyro 475
set_logFrequency, YMagnetometer 293
set_logFrequency, YTilt 376
set_logicalName, YAccelerometer 248
set_logicalName, YCompass 421
set_logicalName, YDataLogger 515
set_logicalName, YGyro 476
set_logicalName, YMagnetometer 294
set_logicalName, YModule 201
set_logicalName, YRefFrame 335

set_logicalName, YTilt 377
set_lowestValue, YAccelerometer 249
set_lowestValue, YCompass 422
set_lowestValue, YGyro 477
set_lowestValue, YMagnetometer 295
set_lowestValue, YTilt 378
set_luminosity, YModule 202
set_mountPosition, YRefFrame 336
set_recording, YDataLogger 516
set_reportFrequency, YAccelerometer 250
set_reportFrequency, YCompass 423
set_reportFrequency, YGyro 478
set_reportFrequency, YMagnetometer 296
set_reportFrequency, YTilt 379
set_resolution, YAccelerometer 251
set_resolution, YCompass 424
set_resolution, YGyro 479
set_resolution, YMagnetometer 297
set_resolution, YTilt 380
set_timeUTC, YDataLogger 517
set_userData, YAccelerometer 252
set_userData, YCompass 425
set_userData, YDataLogger 518
set_userData, YGyro 480
set_userData, YMagnetometer 298
set_userData, YModule 203
set_userData, YRefFrame 337
set_userData, YTilt 381
set_userVar, YModule 204
SetDelegate, YAPI 151
SetTimeout, YAPI 152
Sleep, YAPI 153
Source 93
Start 33
start3DCalibration, YRefFrame 338

T

Test 11
Tilt 28, 36, 39, 47, 57, 65, 72, 80, 87, 93, 99, 108, 341
triggerFirmwareUpdate, YModule 205
TriggerHubDiscovery, YAPI 154
Troubleshooting 541

U

UnregisterHub, YAPI 155
Unsupported 117
UpdateDeviceList, YAPI 156
UpdateDeviceList_async, YAPI 157
updateFirmware, YModule 206

V

Value 534
Variants 63
VirtualHub 105, 117
Visual 71, 79

W

wait_async, YAccelerometer 253
wait_async, YCompass 426
wait_async, YDataLogger 519
wait_async, YGyro 481
wait_async, YMagnetometer 299
wait_async, YModule 207
wait_async, YRefFrame 339
wait_async, YTilt 382

Y

YAccelerometer 211-253
YAPI 135-157
yCheckLogicalName 135
YCompass 386-426
YDataLogger 485-519
YDataSet 522-532
yDisableExceptions 136
yEnableExceptions 137
yEnableUSBHost 138
yFindAccelerometer 211
yFindCompass 386
yFindDataLogger 485
yFindGyro 430
yFindMagnetometer 257
yFindModule 161
yFindRefFrame 303
yFindTilt 343
yFirstAccelerometer 212
yFirstCompass 387
yFirstDataLogger 486
yFirstGyro 431
yFirstMagnetometer 258
yFirstModule 162
yFirstRefFrame 304
yFirstTilt 344
yFreeAPI 139
yGetAPIVersion 140
yGetTickCount 141
YGyro 430-481
yHandleEvents 142
yInitAPI 143
YMagnetometer 257-299
YMeasure 534-538
YModule 161-207
Yocto-3D 7, 20, 35, 39, 47, 57, 65, 71, 79, 87, 93, 99, 105
yPreregisterHub 144
YRefFrame 303-339
yRegisterDeviceArrivalCallback 145
yRegisterDeviceRemovalCallback 146
yRegisterHub 147
yRegisterHubDiscoveryCallback 148
yRegisterLogFunction 149
ySelectArchitecture 150
ySetDelegate 151
ySetTimeout 152

ySleep 153
YTilt 343-382
yTriggerHubDiscovery 154

yUnregisterHub 155
yUpdateDeviceList 156
yUpdateDeviceList_async 157