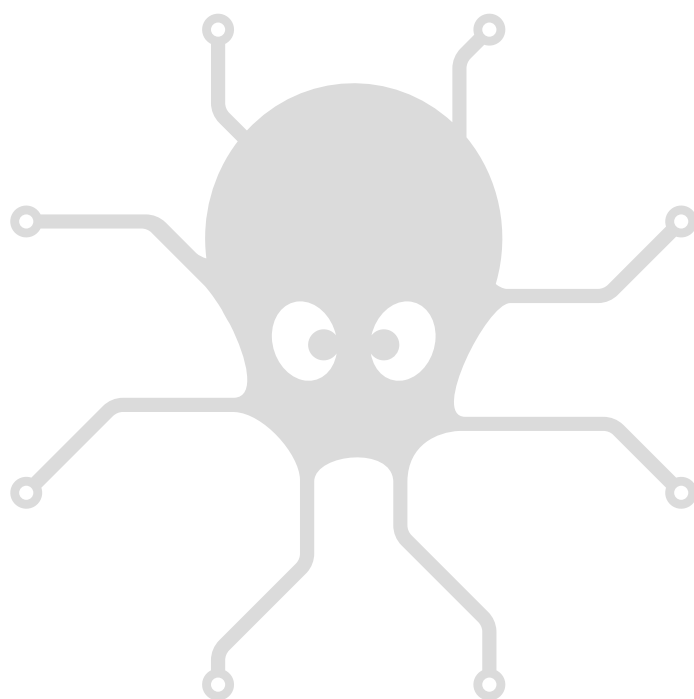




Yocto-CO2

User's guide

Table of contents

1. Introduction	1
1.1. Safety Information	2
1.2. Environmental conditions	2
2. Presentation	3
2.1. Common elements	3
2.2. Specific elements	4
2.3. Optional accessories	4
3. First steps	9
3.1. Prerequisites	9
3.2. Testing USB connectivity	10
3.3. Localization	11
3.4. Test of the module	11
3.5. Configuration	11
4. Assembly and connections	13
4.1. Fixing	13
4.2. USB power distribution	13
5. Programming, general concepts	15
5.1. Programming paradigm	15
5.2. The Yocto-CO2 module	17
5.3. Module control interface	18
5.4. CarbonDioxide function interface	19
5.5. DataLogger function interface	20
5.6. What interface: Native, DLL or Service ?	21
5.7. Programming, where to start?	23
6. Using the Yocto-CO2 in command line	25
6.1. Installing	25
6.2. Use: general description	25
6.3. Control of the CarbonDioxide function	26
6.4. Control of the module part	26

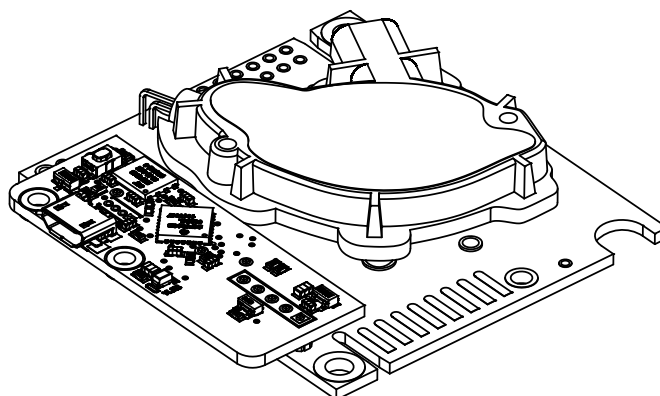
6.5. Limitations	27
7. Using Yocto-CO2 with JavaScript / EcmaScript	29
7.1. Blocking I/O versus Asynchronous I/O in JavaScript	29
7.2. Using Yoctopuce library for JavaScript / EcmaScript 2017	30
7.3. Control of the CarbonDioxide function	32
7.4. Control of the module part	35
7.5. Error handling	37
8. Using Yocto-CO2 with PHP	39
8.1. Getting ready	39
8.2. Control of the CarbonDioxide function	39
8.3. Control of the module part	41
8.4. HTTP callback API and NAT filters	44
8.5. Error handling	47
9. Using Yocto-CO2 with C++	49
9.1. Control of the CarbonDioxide function	49
9.2. Control of the module part	51
9.3. Error handling	54
9.4. Integration variants for the C++ Yoctopuce library	54
10. Using Yocto-CO2 with Objective-C	57
10.1. Control of the CarbonDioxide function	57
10.2. Control of the module part	59
10.3. Error handling	61
11. Using Yocto-CO2 with Visual Basic .NET	63
11.1. Installation	63
11.2. Using the Yoctopuce API in a Visual Basic project	63
11.3. Control of the CarbonDioxide function	64
11.4. Control of the module part	66
11.5. Error handling	68
12. Using Yocto-CO2 with C#	69
12.1. Installation	69
12.2. Using the Yoctopuce API in a Visual C# project	69
12.3. Control of the CarbonDioxide function	70
12.4. Control of the module part	72
12.5. Error handling	74
13. Using the Yocto-CO2 with Universal Windows Platform	77
13.1. Blocking and asynchronous functions	77
13.2. Installation	78
13.3. Using the Yoctopuce API in a Visual Studio project	78
13.4. Control of the CarbonDioxide function	79
13.5. A real example	80
13.6. Control of the module part	80
13.7. Error handling	83
14. Using Yocto-CO2 with Delphi	85
14.1. Preparation	85

14.2. Control of the CarbonDioxide function	85
14.3. Control of the module part	87
14.4. Error handling	89
15. Using the Yocto-CO2 with Python	91
15.1. Source files	91
15.2. Dynamic library	91
15.3. Control of the CarbonDioxide function	91
15.4. Control of the module part	93
15.5. Error handling	95
16. Using the Yocto-CO2 with Java	97
16.1. Getting ready	97
16.2. Control of the CarbonDioxide function	97
16.3. Control of the module part	99
16.4. Error handling	101
17. Using the Yocto-CO2 with Android	103
17.1. Native access and VirtualHub	103
17.2. Getting ready	103
17.3. Compatibility	103
17.4. Activating the USB port under Android	104
17.5. Control of the CarbonDioxide function	106
17.6. Control of the module part	108
17.7. Error handling	113
18. Advanced programming	115
18.1. Event programming	115
18.2. The data logger	118
18.3. Sensor calibration	120
19. Firmware Update	125
19.1. The VirtualHub or the YoctoHub	125
19.2. The command line library	125
19.3. The Android application Yocto-Firmware	125
19.4. Updating the firmware with the programming library	126
19.5. The "update" mode	128
20. Using with unsupported languages	129
20.1. Command line	129
20.2. VirtualHub and HTTP GET	129
20.3. Using dynamic libraries	131
20.4. Porting the high level library	134
21. High-level API Reference	135
21.1. General functions	136
21.2. Module control interface	171
21.3. CarbonDioxide function interface	237
21.4. DataLogger function interface	298
21.5. Recorded data sequence	342
21.6. Measured value	356

22. Troubleshooting	363
22.1. <i>Where to start?</i>	363
22.2. <i>Programming examples don't seem to work</i>	363
22.3. <i>Linux and USB</i>	363
22.4. <i>ARM Platforms: HF and EL</i>	364
22.5. <i>Powered module but invisible for the OS</i>	364
22.6. <i>Another process named xxx is already using yAPI</i>	364
22.7. <i>Disconnections, erratic behavior</i>	364
22.8. <i>Damaged device</i>	365
23. Characteristics	367

1. Introduction

The Yocto-CO2 module is a roughly 58x57mm module which allows you to measure by USB the CO2 rate present in the air up to 10000 ppm. It has a 30 ppm 5% accuracy. The Yocto-CO2 is based on a SenseAir K30 module.



The Yocto-CO2 module

The Yocto-CO2 is not in itself a complete product. It is a component intended to be integrated into a solution used in laboratory equipments, or in industrial process-control equipments, or for similar applications in domestic and commercial environments. In order to use it, you must at least install it in a protective enclosure and connect it to a host computer.

Yoctopuce thanks you for buying this Yocto-CO2 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-CO2 is easy to install anywhere and easy to drive from a maximum of programming languages. If you are nevertheless disappointed with this module, or if you need additional information, do not hesitate to contact Yoctopuce support:

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

1.1. Safety Information

The Yocto-CO2 is designed to meet the requirements of IEC 61010-1:2010 safety standard. It does not create any serious hazards to the operator and surrounding area, even in single fault condition, as long as it is integrated and used according to the instructions contained in this documentation, and in this section in particular.

Protective enclosure

The Yocto-CO2 should not be used without a protective enclosure, because of the accessible bare electronic components. For optimal safety, it should be put into a non-metallic, non-inflammable enclosure, resistant to a mechanical stress level of 5 J. For instance, use a polycarbonate (e.g. LEXAN) enclosure rated IK08 with a IEC 60695-11-10 flammability rating of V-1 or better. Using a lower quality enclosure may require specific warnings for the operator and/or compromise conformity with the safety standard.

Maintenance

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

Identification

In order to ease the maintenance and the identification of risks during maintenance, you should affixate the water-resistant identification label provided together with the electronic board as close as possible to the device. If the device is put in a dedicated enclosure, the identification label should be affixed on the outside of the enclosure.

Application

The safety standard applied is intended to cover laboratory equipment, industrial process-control equipment and similar applications in residential or commercial environment. If you intend to use the Yocto-CO2 for another kind of application, you should check the safety regulations according to the standard applicable to your application.

In particular, the Yocto-CO2 is *not* certified for use in medical environments or for life-support applications.

Environment

The Yocto-CO2 is *not* certified for use in hazardous locations, explosive environments, or life-threatening applications. Environmental ratings are provided below.

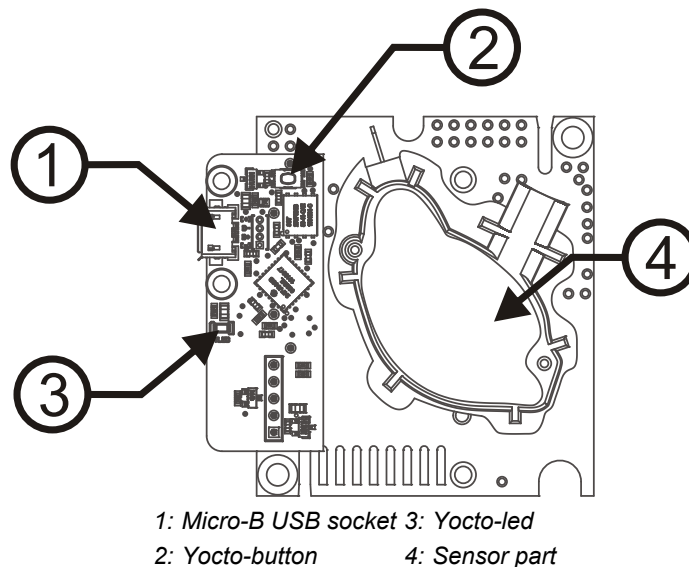
1.2. Environmental conditions

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

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

The extended operating temperature range for the Yocto-CO2 is 0...50°C. This temperature range has been determined based on components manufacturer recommendations, and on controlled environment tests performed during a limited duration (1h). If you plan to use the Yocto-CO2 in harsh environments for a long period of time, we strongly advise you to run extensive tests before going to production.

2. Presentation



2.1. Common elements

All Yocto-modules share a number of common functionalities.

USB connector

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

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

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

Yocto-button

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

Yocto-led

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

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

The Yocto-led has a third functionality, which is less pleasant: when the internal software which controls the module encounters a fatal error, the Yocto-led starts emitting an SOS in morse ¹. If this happens, unplug and re-plug the module. If it happens again, check that the module contains the latest version of the firmware, and, if it is the case, contact Yoctopuce 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-CO2 modules, this number starts with YCO2MK01. 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 Yocto-CO2 is based on a K30 sensor from the Swedish firm SenseAir. Its use does not require any particular precaution.

2.3. Optional accessories

The accessories below are not necessary to use the Yocto-CO2 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.

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

² support@yoctopuce.com

Screws and spacers

In order to mount the Yocto-CO2 module, you can put small screws in the 3mm assembly holes, with a screw head no larger than 8mm. The best way is to use threaded spacers, which you can then mount wherever you want. You can find more details on this topic in the chapter about assembly and connections.

Micro-USB hub

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

YoctoHub-Ethernet, YoctoHub-Wireless and YoctoHub-GSM

You can add network connectivity to your Yocto-CO2, thanks to the YoctoHub-Ethernet, the YoctoHub-Wireless and the YoctoHub-GSM which provides respectively Ethernet, WiFi and GSM connectivity. All of them can drive up to three devices and behave exactly like a regular computer running a *VirtualHub*.

1.27mm (or 1.25mm) connectors

In case you wish to connect your Yocto-CO2 to a Micro-hub USB or a YoctoHub without using a bulky USB connector, you can use the four 1.27mm pads just behind the USB connector. There are two options.

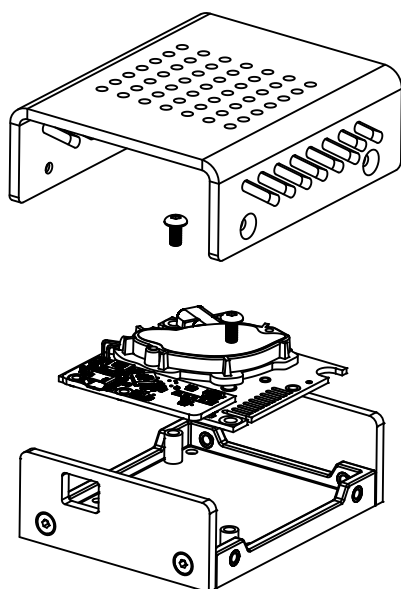
You can mount the Yocto-CO2 directly on the hub using screw and spacers, and connect it using 1.27mm board-to-board connectors. To prevent shortcuts, it is best to solder the female connector on the hub and the male connector on the Yocto-CO2.

You can also use a small 4-wires cable with a 1.27mm connector. 1.25mm works as well, it does not make a difference for 4 pins. This makes it possible to move the device a few inches away. Don't put it too far away if you use that type of cable, because as the cable is not shielded, it may cause undesirable electromagnetic emissions.

Enclosure

Your Yocto-CO2 has been designed to be installed as is in your project. Nevertheless, Yoctopuce sells enclosures specifically designed for Yoctopuce devices. More details are available on the Yoctopuce web site ³. The suggested enclosure model for your Yocto-CO2 is the YoctoBox-CO2-Black.

³ <http://www.yoctopuce.com/EN/products/category/enclosures>



3. First steps

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

3.1. Prerequisites

In order to use your Yocto-CO2 module, you should have the following items at hand.

A computer

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

Yoctopuce provides software libraries to drive its modules for the following operating systems: Windows, macOS X, Linux, and Android. Yoctopuce modules do not require installing any specific system driver, as they leverage the standard HID driver¹ provided with every operating system.

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

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

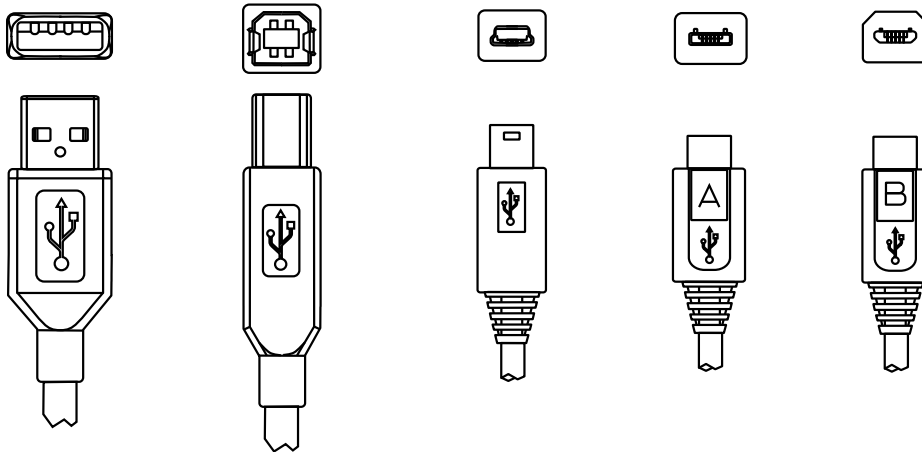
Linux kernels currently supported are the 2.6 branch, the 3.x branch and the 4.x branch. Other versions of the Linux kernel, and even other UNIX variants, are very likely to work as well, as Linux support is implemented through the standard **libusb** API. Yoctopuce is frequently testing its modules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

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

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

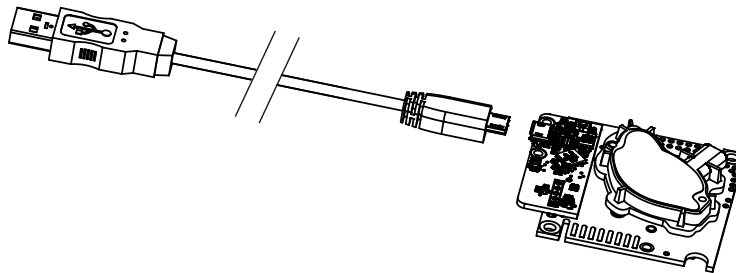
A USB 2.0 cable, type A-micro B

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



The most common USB 2.0 connectors: A, B, Mini B, Micro A, Micro B²

To connect your Yocto-CO2 module to a computer, you need a USB 2.0 cable of type A-micro B. The price of this cable may vary a lot depending on the source, look for it under the name *USB 2.0 A to micro B Data cable*. Make sure not to buy a simple USB charging cable without data connectivity. The correct type of cable is available on the Yoctopuce shop.



You must plug in your Yocto-CO2 module with a USB 2.0 cable of type A - micro B

If you insert a USB hub between the computer and the Yocto-CO2 module, make sure to take into account the USB current limits. If you do not, be prepared to face unstable behaviors and unpredictable failures. You can find more details on this topic in the chapter about assembly and connections.

3.2. Testing USB connectivity

At this point, your Yocto-CO2 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* software³. 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 browser⁴. Start the *Virtual Hub* software in a command line, open your preferred web browser and enter the URL <http://127.0.0.1:4444>. The list of the Yoctopuce modules connected to your computer is displayed.

² Although they existed for some time, Mini A connectors are not available anymore http://www.usb.org/developers/Deprecation_Announcement_052507.pdf

³ www.yoctopuce.com/EN/virtualhub.php

⁴ The interface is tested on Chrome, FireFox, Safari, Edge et IE 11.

Serial	Logical Name	Description	Action
VIRTHUB0-7d1a86fb0		VirtualHub	configure view log file
YCO2MK01-05B8A		Yocto-CO2	configure view log file beacon

[Show device functions](#)

Module list as displayed in your web browser

3.3. Localization

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

3.4. Test of the module

The first item to check is that your module is working well: click on the serial number corresponding to your module. This displays a window summarizing the properties of your Yocto-CO2.

YCO2MK01-05B8A


YCO2MK01-05B8A is a 58x57mm hybrid board featuring a CO2 sensor.

Kernel

Serial # YCO2MK01-05B8A
Product name: Yocto-CO2
Logical name:
Product release: 1
Firmware: 7728
Consumption: 42 mA
Beacon: Inactive [turn on](#)
Luminosity: 50%

Sensors

	Min	Current	Max
CO2	0 ppm	2811 ppm	2860 ppm

CO2 data are shown in ppm by volume.

Misc

[Open API browser \(pop-up\)](#)
[Get user manual from yoctopuce.com](#)

Close

Properties of the Yocto-CO2 module

This window allows you, among other things, to play with your module to check how it is working, as CO2 rate values are displayed in real time.

3.5. Configuration

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

Yocto-CO2 module configuration.

Firmware

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

To update a firmware, simply click on the **upgrade** button on the configuration window and follow the instructions. If the update fails for one reason or another, unplug and re-plug the module and start the update process again. This solves the issue in most cases. If the module was unplugged while it was being reprogrammed, it does probably not work anymore and is not listed in the interface. However, it is always possible to reprogram the module correctly by using the *Virtual Hub* software⁵ in command line⁶.

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 only function provided by the Yocto-CO2 module is the "CarbonDioxide" function. Simply click on the corresponding "rename" button to assign it a new logical name.

⁵ www.yoctopuce.com/EN/virtualhub.php

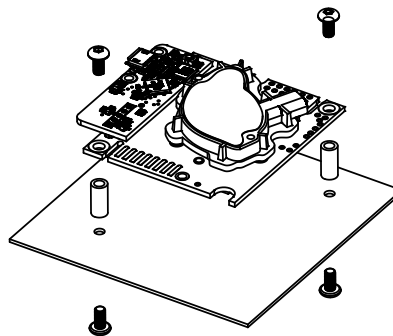
⁶ More information available in the virtual hub documentation

4. Assembly and connections

This chapter provides important information regarding the use of the Yocto-CO2 module in real-world situations. Make sure to read it carefully before going too far into your project if you want to avoid pitfalls.

4.1. Fixing

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



Examples of assembly on supports

The Yocto-CO2 module contains 3mm assembly holes. You can use these holes for screws. We recommend using spacers to fix your module at some distance from the support, in order to avoid damaging the components present below the sensor. Do not use more than two screw/spacer couples to fix your module to minimize mechanical constraints during temperature changes.

4.2. USB power distribution

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

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

cascaded hubs without external power supply. In order to cascade hubs, it is necessary to use self-powered USB hubs, that provide a full 500mA to each subport.

In practice, USB would not have been as successful if it was really so picky about power distribution. As it happens, most USB hub manufacturers have been doing savings by not implementing current limitation on ports: they simply connect the computer power supply to every port, and declare themselves as *self-powered hub* even when they are taking all their power from the USB bus (in order to prevent any power consumption check in the operating system). This looks a bit dirty, but given the fact that computer USB ports are usually well protected by a hardware current limitation around 2000mA, it actually works in every day life, and seldom makes hardware damage.

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

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

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

4.3. Electromagnetic compatibility (EMI)

Connection methods to integrate the Yocto-CO2 obviously have an impact on the system overall electromagnetic emissions, and therefore also impact the conformity with international standards.

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

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

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

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

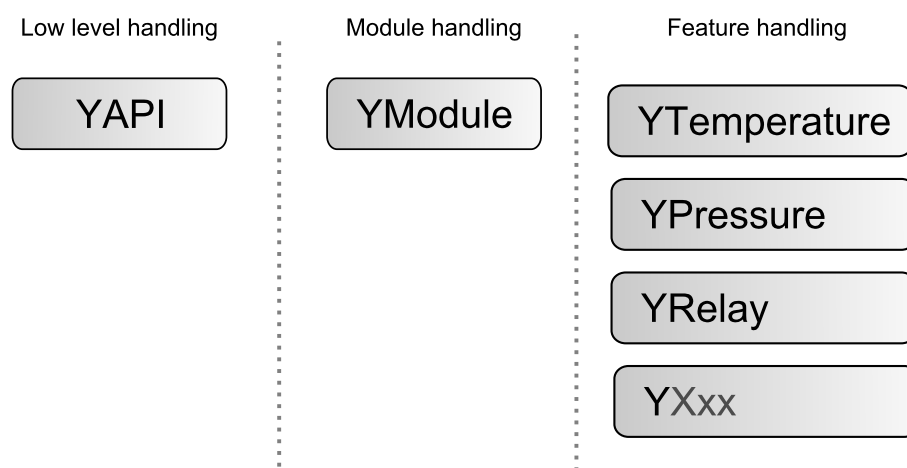
5. Programming, general concepts

The Yoctopuce API was designed to be at the same time simple to use and sufficiently generic for the concepts used to be valid for all the modules in the Yoctopuce range, and this in all the available programming languages. Therefore, when you have understood how to drive your Yocto-CO2 with your favorite programming language, learning to use another module, even with a different language, will most likely take you only a minimum of time.

5.1. Programming paradigm

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

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



Structure of the Yoctopuce API.

The YSensor class

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

The YSensor class is the parent class for all Yoctopuce sensors, and can provide access to any sensor, regardless of its type. It includes methods to access all common functions. This makes it easier to create applications that use many different sensors. Moreover, if you create an application based on YSensor, it will work with all Yoctopuce sensors, even those which do not yet exist.

Programmation

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

From the programming stand point, your Yocto-CO2 is viewed as a module hosting a given number of functions. In the API, these functions are objects which can be found independently, in several ways.

Access to the functions of a module

Access by logical name

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

Access by enumeration

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

Access by hardware name

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

Difference between *Find* and *First*

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

Function handling

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

Access to the modules

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

Functions/Module interaction

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

5.2. The Yocto-CO2 module

The Yocto-CO2 module provides a single instance of the carbonDioxide and dataLogger functions, corresponding to the CO2 sensor and to the data logger.

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

carbonDioxide : CarbonDioxide

attribute	type	modifiable ?
logicalName	String	modifiable
advertisedValue	String	modifiable
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
advMode	Enumerated	modifiable
calibrationParam	Calibration parameters	modifiable
resolution	Fixed-point number	modifiable
sensorState	Integer	read-only
abcPeriod	Integer	modifiable
command	String	modifiable

dataLogger : DataLogger

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

5.3. 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-CO2 module, this serial number always starts with YCO2MK01. 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 39 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.

5.4. CarbonDioxide function interface

The Yoctopuce class YCarbonDioxide allows you to read and configure Yoctopuce CO2 sensors. It inherits from YSensor class the core functions to read measurements, to register callback functions, to access the autonomous datalogger. This class adds the ability to perform manual calibration if required.

logicalName

Character string containing the logical name of the CO2 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 CO2 sensor directly. If two CO2 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 CO2 sensor, that is automatically advertised up to the parent hub. For a CO2 sensor, the advertised value is the current value of the CO2 concentration.

unit

Short character string representing the measuring unit for the CO2 concentration.

currentValue

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

lowestValue

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

highestValue

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

currentRawValue

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

logFrequency

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

reportFrequency

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

advMode

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

calibrationParam

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

resolution

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

sensorState

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

abcPeriod

Automatic baseline calibration period, in hours. A negative value corresponds to disabled automatic baseline calibration.

command

Magic attribute used to setup physical sensor parameters.

5.5. 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. When the datalogger is enabled but not yet ready to record data, its state is set to PENDING.

autoStart

Automatic start of the data logger on power on. Setting this attribute ensures that the data logger is always turned on when the device is powered up, without need for a software command.

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

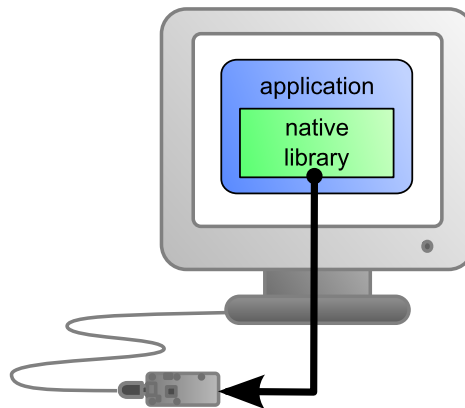
Attribute that can be set to true to clear recorded data.

5.6. What interface: Native, DLL or Service ?

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

Native control

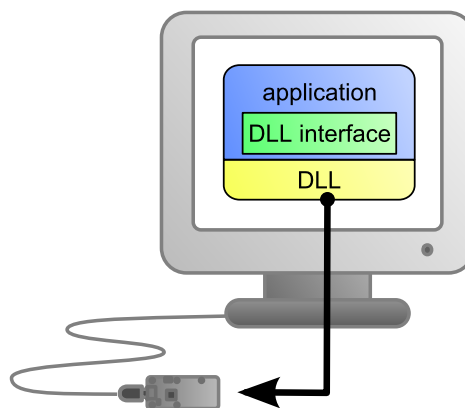
In this case, the software driving your project is compiled directly with a library which provides control of the modules. Objectively, it is the simplest and most elegant solution for the end user. The end user then only needs to plug the USB cable and run your software for everything to work. Unfortunately,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



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

Native control by DLL

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



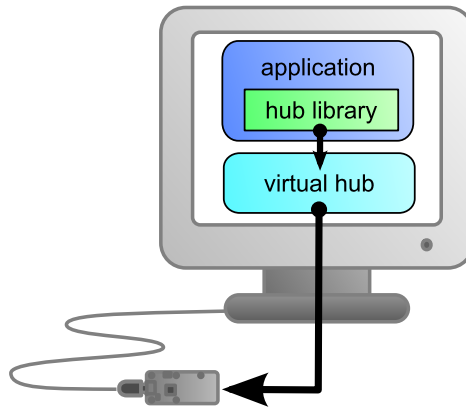
The application uses the DLL to natively control the locally connected module

Control by service

Some languages do simply not allow you to easily gain access to the hardware layers of the machine. It is the case for Javascript, for instance. To deal with this case, Yoctopuce provides a solution in the form of a small piece of software called *VirtualHub*¹. It can access the modules, and your application only needs to use a library which offers all necessary functions to control the modules via this VirtualHub. The end users will have to start the VirtualHub before running the

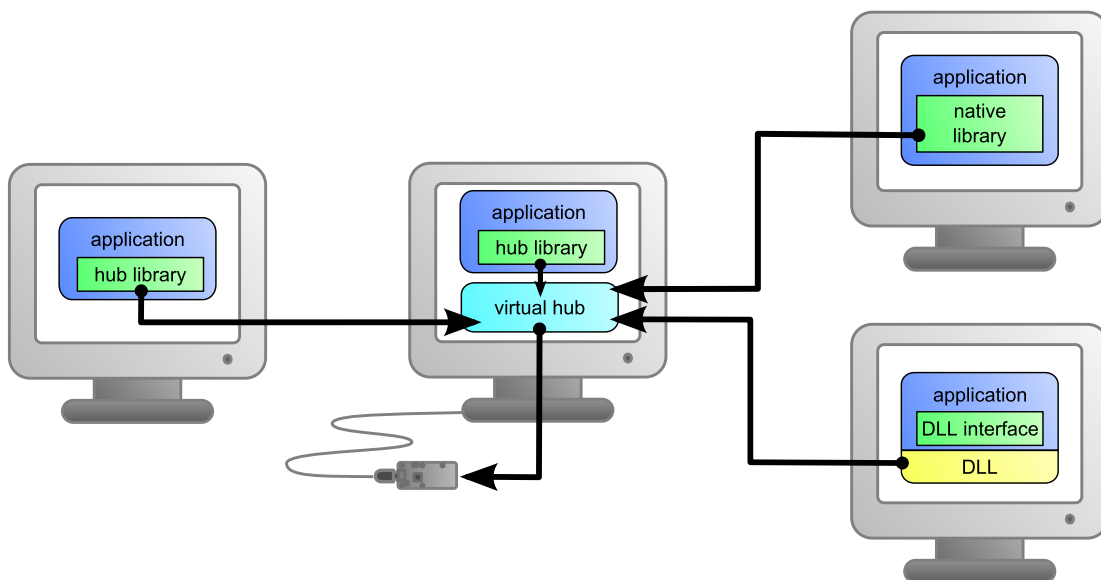
¹ www.yoctopuce.com/EN/virtualhub.php

project control software itself, unless they decide to install the hub as a service/deamon, in which case the VirtualHub starts automatically when the machine starts up.



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

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



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

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

Language	Native	Native with DLL	Virtual hub
C++	✓	✓	✓
Objective-C	✓	-	✓
Delphi	-	✓	✓
Python	-	✓	✓
VisualBasic .Net	-	✓	✓
C# .Net	-	✓	✓
C# UWP	✓	-	✓
EcmaScript / JavaScript	-	-	✓
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.

5.7. Programming, where to start?

At this point of the user's guide, you should know the main theoretical points of your Yocto-CO2. 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-CO2.

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

6. Using the Yocto-CO2 in command line

When you want to perform a punctual operation on your Yocto-CO2, 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.

6.1. 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-CO2, open a shell, and start working by typing for example:

```
C:\>YCarbonDioxide 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.

6.2. Use: general description

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

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

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

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

¹ If you want to recompile the command line API, you also need the C++ API.

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

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

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

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

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

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

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

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

6.3. Control of the CarbonDioxide function

To control the CarbonDioxide function of your Yocto-CO2, you need the YCarbonDioxide executable file.

For instance, you can launch:

```
C:\>YCarbonDioxide any get_currentValue
```

This example uses the "any" target to indicate that we want to work on the first CarbonDioxide 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-CO2 module with the *YCO2MK01-123456* serial number which you have called "*MyModule*", and its carbonDioxide 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:\>YCarbonDioxide YCO2MK01-123456.carbonDioxide describe
C:\>YCarbonDioxide YCO2MK01-123456.MyFunction describe
C:\>YCarbonDioxide MyModule.carbonDioxide describe
C:\>YCarbonDioxide MyModule.MyFunction describe
C:\>YCarbonDioxide MyFunction describe
```

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

```
C:\>YCarbonDioxide all describe
```

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

```
C:\>YCarbonDioxide /help
```

6.4. Control of the module part

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

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


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

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

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

```
C:\>YModule YCO2MK01-12346 set_logicalName MonPremierModule
C:\>YModule YCO2MK01-12346 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 YCO2MK01-12346 set_logicalName MonPremierModule
C:\>YModule YCO2MK01-12346 saveToFlash
```

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

```
C:\>YModule -s YCO2MK01-12346 set_logicalName MonPremierModule
```

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

6.5. Limitations

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

You can easily work around this limitation by using a Virtual Hub: run the 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>

7. Using Yocto-CO2 with JavaScript / EcmaScript

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.2. Using Yoctopuce library for JavaScript / EcmaScript 2017

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

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

- The Javascript / EcmaScript 2017 programming library¹
- The VirtualHub software² for Windows, Mac OS X or Linux, depending on your OS

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

Using the official Yoctopuce library for node.js

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

¹ www.yoctopuce.com/EN/libraries.php

² www.yoctopuce.com/EN/virtualhub.php

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

```
npm install
```

Once done, you can start the example file using:

```
node demo.js
```

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

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

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

Using the Yoctopuce library within a browser (HTML)

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

Using the Yoctopuce library on older JavaScript engines

If you need to run this library on older JavaScript engines, you can use Babel³ to transpile your code and the library into older JavaScript standards. To install Babel with typical settings, simply use:

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

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

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

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

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

Backward-compatibility with the old JavaScript library

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

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

by

³ <http://babeljs.io>

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

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

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

by

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

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

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

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

you can use:

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

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

You can also rewrite this last asynchronous call as:

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

7.3. Control of the CarbonDioxide function

A few lines of code are enough to use a Yocto-CO2. Here is the skeleton of a JavaScript code snippet to use the CarbonDioxide function.

```
// For Node.js, we use function require()
// For HTML, we would use <script src="...">
require('yoctolib-es2017/yocto_api.js');
require('yoctolib-es2017/yocto_carbondioxide.js');

// Get access to your device, through the VirtualHub running locally
await YAPI.RegisterHub('127.0.0.1');
var carbondioxide = YCarbonDioxide.FindCarbonDioxide("YCO2MK01-123456.carbonDioxide");

// Check that the module is online to handle hot-plug
if(await carbondioxide.isOnline())
{
  // Use carbondioxide.get_currentValue()
  [...]
```

```
}
```

Let us look at these lines in more details.

yocto_api and yocto_carbondioxide import

These two imports provide access to functions allowing you to manage Yoctopuce modules. `yocto_api` is always needed, `yocto_carbondioxide` is necessary to manage modules containing a CO2 sensor, such as Yocto-CO2. Other imports can be useful in other cases, such as `YModule` which can let you enumerate any type of Yoctopuce device.

YAPI.RegisterHub

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

YCarbonDioxide.FindCarbonDioxide

The `FindCarbonDioxide` method allows you to find a CO2 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-CO2 module with serial number `YCO2MK01-123456` which you have named `"MyModule"`, and for which you have given the `carbonDioxide` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
carbonDioxide = YCarbonDioxide.FindCarbonDioxide("YCO2MK01-123456.carbonDioxide")
carbonDioxide = YCarbonDioxide.FindCarbonDioxide("YCO2MK01-123456.MaFonction")
carbonDioxide = YCarbonDioxide.FindCarbonDioxide("MonModule.carbonDioxide")
carbonDioxide = YCarbonDioxide.FindCarbonDioxide("MonModule.MaFonction")
carbonDioxide = YCarbonDioxide.FindCarbonDioxide("MaFonction")
```

`YCarbonDioxide.FindCarbonDioxide` returns an object which you can then use at will to control the CO2 sensor.

isOnline

The `isOnline()` method of the object returned by `FindCarbonDioxide` 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 `YCarbonDioxide.FindCarbonDioxide` provides the CO2 level currently measured by the sensor. The value returned is a floating number in parts per million (volumic)

A real example, for Node.js

Open a command window (a terminal, a shell...) and go into the directory **example_nodejs/Doc-GettingStarted-Yocto-CO2** within Yoctopuce library for JavaScript / EcmaScript 2017. In there, you will find a file named `demo.js` with the sample code below, which uses the functions explained above, but this time used with all side materials needed to make it work nicely as a small demo.

If your Yocto-CO2 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-CO2 is connected and where you run the VirtualHub.

```
"use strict";

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

let co2;
```

```

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

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

  // Select specified device, or use first available one
  let serial = process.argv[process.argv.length-1];
  if(serial[8] !== '-') {
    // by default use any connected module suitable for the demo
    let anysensor = YCarbonDioxide.FirstCarbonDioxide();
    if(anysensor) {
      let module = await anysensor.module();
      serial = await module.get_serialNumber();
    } else {
      console.log('No matching sensor connected, check cable !');
      return;
    }
  }
  console.log('Using device '+serial);
  co2 = YCarbonDioxide.FindCarbonDioxide(serial+'.carbonDioxide');

  refresh();
}

async function refresh()
{
  if (await co2.isOnline()) {
    console.log('Carbon Dioxide : '+(await co2.get_currentValue()) + ' ppm');
  } else {
    console.log('Module not connected');
  }
  setTimeout(refresh, 500);
}

startDemo();

```

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

```
npm install
```

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

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

Same example, but this time running in a browser

If you want to see how to use the library within a browser rather than with Node.js, switch to the directory **example_html/Doc-GettingStarted-Yocto-CO2**. You will find there a single HTML file, with a JavaScript section similar to the code above, but with a few changes since it has to interact through an HTML page rather than through the JavaScript console.

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Hello World</title>
  <script src="../../lib/yocto_api.js"></script>
  <script src="../../lib/yocto_carbondioxide.js"></script>
  <script>
    async function startDemo()
    {

```



```

    await YAPI.LogUnhandledPromiseRejections();
    await YAPI.DisableExceptions();

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

async function refresh()
{
    let serial = document.getElementById('serial').value;
    if(serial == '') {
        // by default use any connected module suitable for the demo
        let anysensor = YCarbonDioxide.FirstCarbonDioxide();
        if(anysensor) {
            let module = await anysensor.module();
            serial = await module.get_serialNumber();
            document.getElementById('serial').value = serial;
        }
    }
    let co2 = YCarbonDioxide.FindCarbonDioxide(serial+'.carbonDioxide');

    if (await co2.isOnline()) {
        document.getElementById('msg').value = '';
        document.getElementById("co2").value = (await co2.get_currentValue()) + ' ppm';
    } else {
        document.getElementById('msg').value = 'Module not connected';
    }
    setTimeout(refresh, 500);
}

startDemo();
</script>
</head>
<body>
Module to use: <input id='serial'>
<input id='msg' style='color:red;border:none;' readonly><br>
Carbon dioxide : <input id='co2' readonly><br>
</body>
</html>

```

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

7.4. Control of the module part

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

```

"use strict";

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

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

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

    // Select the relay to use
    let module = YModule.FindModule(args[0]);
    if(await module.isOnline()) {
        if(args.length > 1) {
            if(args[1] == 'ON') {
                await module.set_beacon(YModule.BEACON_ON);
            } else {

```

```

        await module.set_beacon(YModule.BEACON_OFF);
    }
    console.log('serial:      '+await module.get_serialNumber());
    console.log('logical name: '+await module.get_logicalName());
    console.log('luminosity:   '+await module.get_luminosity()+'%');
    console.log('beacon:      '+ (await module.get_beacon() == YModule.BEACON_ON
?'ON':'OFF'));
    console.log('upTime:      '+parseInt(await module.get_upTime()/1000)+' sec');
    console.log('USB current:  '+await module.get_usbCurrent()+' mA');
    console.log('logs:');
    console.log(await module.get_lastLogs());
    } else {
        console.log("Module not connected (check identification and USB cable)\n");
    }
    await YAPI.FreeAPI();
}

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

```

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

Changing the module settings

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

```

"use strict";

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

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

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

    // Select the relay to use
    let module = YModule.FindModule(args[0]);
    if(await module.isOnline()) {
        if(args.length > 1) {
            let newname = args[1];
            if (!await YAPI.CheckLogicalName(newname)) {
                console.log("Invalid name (" + newname + ")");
                process.exit(1);
            }
            await module.set_logicalName(newname);
            await module.saveToFlash();
        }
        console.log('Current name: '+await module.get_logicalName());
    } else {
        console.log("Module not connected (check identification and USB cable)\n");
    }
    await YAPI.FreeAPI();
}

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

```

```

} else {
    startDemo(process.argv.slice(2));
}

```

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

Listing the modules

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

```

"use strict";

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

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

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

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

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

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

```

7.5. Error handling

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

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

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

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

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

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

8. Using Yocto-CO2 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 you machine. If you wish to use the PHP server of your internet provider, it is possible, but you will probably need to configure your ADSL router for it to accept and forward TCP request on the 4444 port.

8.1. Getting ready

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

- The PHP programming 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.

8.2. Control of the CarbonDioxide function

A few lines of code are enough to use a Yocto-CO2. Here is the skeleton of a PHP code snippet to use the CarbonDioxide function.

```
include('yocto_api.php');  
include('yocto_carbondioxide.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/', $errmsg);
$carbondioxide = yFindCarbonDioxide("YCO2MK01-123456.carbonDioxide");

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

Let's look at these lines in more details.

yocto_api.php and yocto_carbondioxide.php

These two PHP includes provides access to the functions allowing you to manage Yoctopuce modules. `yocto_api.php` must always be included, `yocto_carbondioxide.php` is necessary to manage modules containing a CO2 sensor, such as Yocto-CO2.

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.

yFindCarbonDioxide

The `yFindCarbonDioxide` function allows you to find a CO2 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-CO2 module with serial number `YCO2MK01-123456` which you have named `"MyModule"`, and for which you have given the `carbonDioxide` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
$carbondioxide = yFindCarbonDioxide("YCO2MK01-123456.carbonDioxide");
$carbondioxide = yFindCarbonDioxide("YCO2MK01-123456.MyFunction");
$carbondioxide = yFindCarbonDioxide("MyModule.carbonDioxide");
$carbondioxide = yFindCarbonDioxide("MyModule.MyFunction");
$carbondioxide = yFindCarbonDioxide("MyFunction");
```

`yFindCarbonDioxide` returns an object which you can then use at will to control the CO2 sensor.

isOnline

The `isOnline()` method of the object returned by `yFindCarbonDioxide` 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 `yFindCarbonDioxide` provides the CO2 level currently measured by the sensor. The value returned is a floating number in parts per million (volumic).

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 you web server, then use your preferred web browser to access this page. The code is also provided in the directory **Examples/Doc-GettingStarted-Yocto-CO2** 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.

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

```

<HTML>
<HEAD>
  <TITLE>Hello World</TITLE>
</HEAD>
<BODY>
<?php
  include('yocto_api.php');
  include('yocto_carbondioxide.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
    $co2 = yFindCarbonDioxide("$serial.carbonDioxide");
    if (!$co2->isOnline()) {
      die("Module not connected (check serial and USB cable)");
    }
  } else {
    // or use any connected module suitable for the demo
    $co2 = yFirstCarbonDioxide();
    if(is_null($co2)) {
      die("No module connected (check USB cable)");
    } else {
      $serial = $co2->module()->get_serialnumber();
    }
  }
  Print("Module to use: <input name='serial' value='$serial'><br>");

  $tvalue = $co2->get_currentValue();
  Print("CO2: $tvalue ppm<br>");
  yFreeAPI();

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

```

8.3. Control of the module part

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

```

<HTML>
<HEAD>
  <TITLE>Module Control</TITLE>
</HEAD>
<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());
    yFreeAPI();
?>
<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>");
yFreeAPI();
?>
<input type='submit'>
</FORM>
</BODY>
</HTML>

```

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

Listing the modules

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

```

<HTML>
<HEAD>
  <TITLE>inventory</TITLE>
</HEAD>
<BODY>
  <H1>Device list</H1>
  <TT>
    <?php
      include('yocto_api.php');
      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();
      }
      yFreeAPI();
    ?>
  </TT>
</BODY>
</HTML>

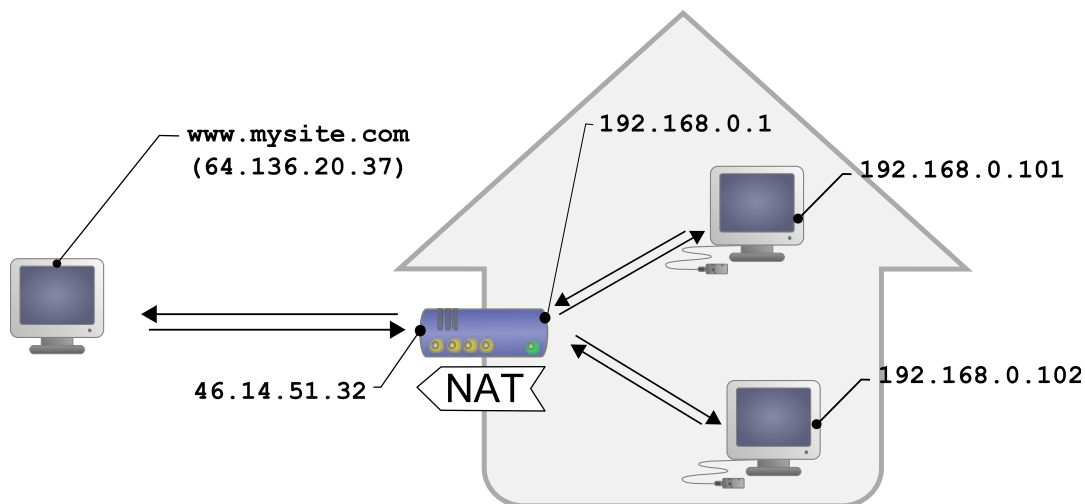
```

8.4. HTTP callback API and NAT filters

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

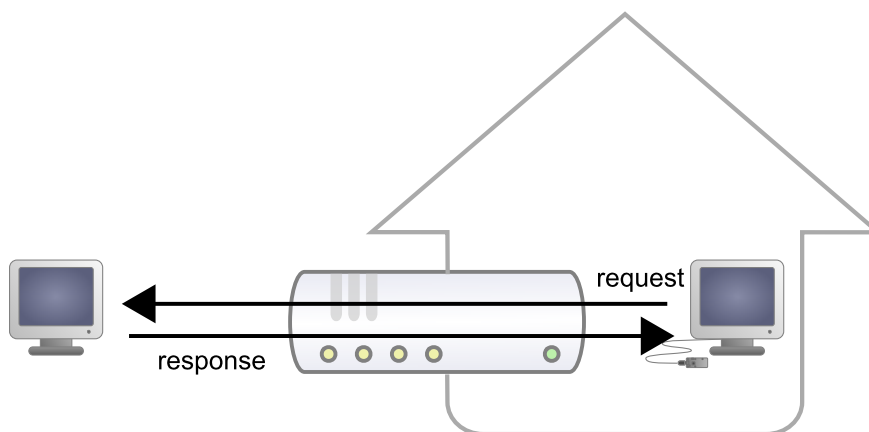
The NAT filter: advantages and disadvantages

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

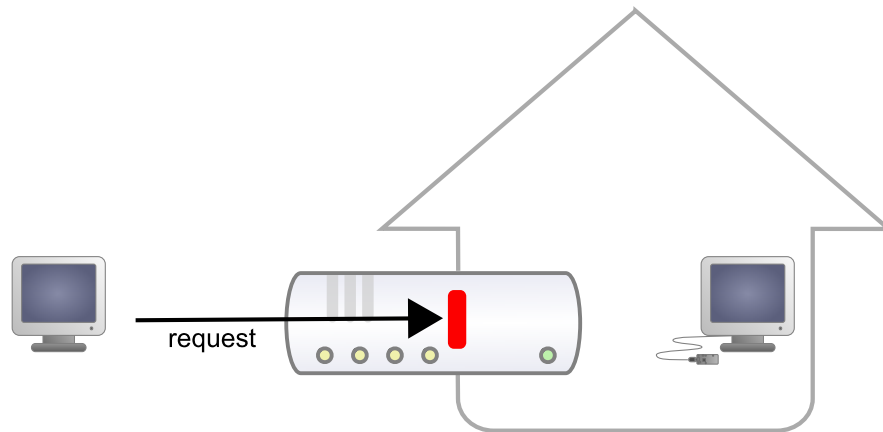


Typical DSL configuration: LAN machines are isolated from the outside by the DSL router

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



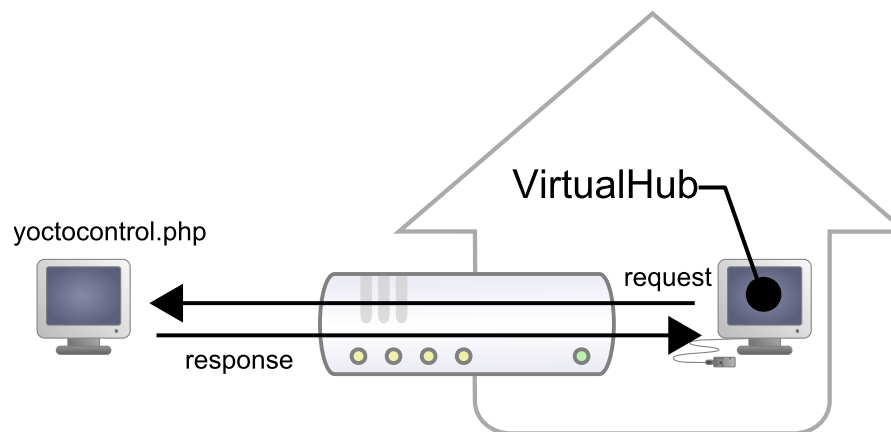
Responses from request from LAN machines are routed.



But requests from the outside are blocked.

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

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



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

Configuration

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

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

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

Serial	Logical Name	Description	Action
VIRIHUB0-7d1a86fb0	VirtualHub	VirtualHub	configure view log file
RELAYHI1-00055	Yocto-PowerRelay	Yocto-PowerRelay	configure view log file beacon
TMPSENS1-05E7F	Yocto-Temperature	Yocto-Temperature	configure view log file beacon

Click on the "configure" button on the first line

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

Serial #: VIRTHUB0-7d1a86fb09
 Product name: VirtualHub
 Software version: 10789
 Logical name:

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: **Yocto-API callback**

Yoctopuce devices can be controlled through remote PHP scripts. That Yocto-API callback protocol is designed so it can pass through NAT filters without opening ports. See your device user manual, *PHP programming* section for more details.

2. Specify the URL to use for reporting values. *HTTPS protocol is not yet supported.*
 Callback URL:

3. If your callback requires authentication, enter credentials here. Digest authentication is recommended, but Basic authentication works as well.
 Username:
 Password:

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, EcmaScript (Node.JS) and Java only.

8.5. Error handling

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

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

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

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

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

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

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

9. Using Yocto-CO2 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 Mac 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.

9.1. Control of the CarbonDioxide function

A few lines of code are enough to use a Yocto-CO2. Here is the skeleton of a C++ code snippet to use the CarbonDioxide function.

```
#include "yocto_api.h"
#include "yocto_carbondioxide.h"

[...]
String errmsg;
YCarbonDioxide *carbondioxide;

// Get access to your device, connected locally on USB for instance
yRegisterHub("usb", errmsg);
carbondioxide = yFindCarbonDioxide("YCO2MK01-123456.carbonDioxide");
```

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(carbondioxide->isOnline())
{
    // Use carbondioxide->get_currentValue(), ...
}
```

Let's look at these lines in more details.

yocto_api.h et yocto_carbondioxide.h

These two include files provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api.h` must always be used, `yocto_carbondioxide.h` is necessary to manage modules containing a CO2 sensor, such as Yocto-CO2.

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.

yFindCarbonDioxide

The `yFindCarbonDioxide` function allows you to find a CO2 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-CO2 module with serial number `YCO2MK01-123456` which you have named `"MyModule"`, and for which you have given the `carbonDioxide` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
YCarbonDioxide *carbondioxide = yFindCarbonDioxide("YCO2MK01-123456.carbonDioxide");
YCarbonDioxide *carbondioxide = yFindCarbonDioxide("YCO2MK01-123456.MyFunction");
YCarbonDioxide *carbondioxide = yFindCarbonDioxide("MyModule.carbonDioxide");
YCarbonDioxide *carbondioxide = yFindCarbonDioxide("MyModule.MyFunction");
YCarbonDioxide *carbondioxide = yFindCarbonDioxide("MyFunction");
```

`yFindCarbonDioxide` returns an object which you can then use at will to control the CO2 sensor.

isOnline

The `isOnline()` method of the object returned by `yFindCarbonDioxide` 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 `yFindCarbonDioxide` provides the CO2 level currently measured by the sensor. The value returned is a floating number in parts per million (volumic).

A real example

Launch your C++ environment and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-CO2** 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_carbondioxide.h"
#include <iostream>
#include <stdlib.h>

using namespace std;
```



```

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

int main(int argc, const char * argv[])
{
    string errmsg, target;
    YCarbonDioxide *co2sensor;

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

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

    if (target == "any") {
        co2sensor = yFirstCarbonDioxide();
        if (co2sensor == NULL) {
            cout << "No module connected (check USB cable)" << endl;
            return 1;
        }
    } else {
        co2sensor = yFindCarbonDioxide(target + ".carbonDioxide");
    }

    while (1) {
        if (!co2sensor->isOnline()) {
            cout << "Module not connected (check identification and USB cable)";
            break;
        }
        cout << "CO2: " << co2sensor->get_currentValue() << " ppm" << endl;
        cout << "  (press Ctrl-C to exit)" << endl;
        ySleep(1000, errmsg);
    };

    yFreeAPI();
    return 0;
}

```

9.2. Control of the module part

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

```

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

#include "yocto_api.h"

using namespace std;

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

int main(int argc, const char * argv[])

```

```

{
    string      errmsg;

    // Setup the API to use local USB devices
    if(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->set_beacon(Y_BEACON_ON);
            else
                module->set_beacon(Y_BEACON_OFF);
        }
        cout << "serial:      " << module->get_serialNumber() << endl;
        cout << "logical name: " << module->get_logicalName() << endl;
        cout << "luminosity:  " << module->get_luminosity() << endl;
        cout << "beacon:      ";
        if (module->get_beacon() == Y_BEACON_ON)
            cout << "ON" << endl;
        else
            cout << "OFF" << endl;
        cout << "upTime:      " << module->get_upTime() / 1000 << " sec" << endl;
        cout << "USB current: " << module->get_usbCurrent() << " mA" << endl;
        cout << "Logs:" << endl << module->get_lastLogs() << endl;
    } else {
        cout << argv[1] << " not connected (check identification and USB cable)"
            << endl;
    }
    yFreeAPI();
    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;
}
yFreeAPI();
return 0;
}

```

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

Listing the modules

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

```

#include <iostream>

#include "yocto_api.h"

using namespace std;

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

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

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

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

```

9.3. Error handling

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

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

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

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

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

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

9.4. Integration variants for the C++ Yoctopuce library

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

Integration in source format

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

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

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

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

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

Integration as a static library

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

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

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

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

Integration as a dynamic library

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

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

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

With GCC, the command line to compile is simply:

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


10. Using Yocto-CO2 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.

10.1. Control of the CarbonDioxide function

Launch Xcode 4.2 and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-CO2** of the Yoctopuce library.

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

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

int main(int argc, const char * argv[])
{
```

¹ www.yoctopuce.com/EN/libraries.php

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

```

NSError *error;

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

@autoreleasepool {
    // Setup the API to use local USB devices
    if([YAPI RegisterHub:@"usb":&error] != YAPI_SUCCESS) {
        NSLog(@"RegisterHub error: %@", [error localizedDescription]);
        return 1;
    }
    NSString *target = [NSString stringWithUTF8String:argv[1]];
    YCarbonDioxide *co2sensor;
    if ([target isEqualToString:@"any"]) {
        co2sensor = [YCarbonDioxide FirstCarbonDioxide];
        if (co2sensor == NULL) {
            NSLog(@"No module connected (check USB cable)");
            return 1;
        }
    } else {
        co2sensor = [YCarbonDioxide FindCarbonDioxide:[target stringByAppendingString:
            @".carbonDioxide"]];
    }

    while(1) {
        if(![co2sensor isOnline]) {
            NSLog(@"Module not connected (check identification and USB cable)\n");
            break;
        }

        NSLog(@"CO2: %f ppm\n", [co2sensor currentValue]);
        NSLog(@"    (press Ctrl-C to exit)\n");
        [YAPI Sleep:1000: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_carbondioxide.h

These two import files provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api.h` must always be used, `yocto_carbondioxide.h` is necessary to manage modules containing a CO2 sensor, such as Yocto-CO2.

[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.

[CarbonDioxide FindCarbonDioxide]

The `[CarbonDioxide FindCarbonDioxide]` function allows you to find a CO2 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-CO2 module with serial number `YCO2MK01-123456` which you have named `"MyModule"`, and for which you have given the `carbonDioxide` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```

YCarbonDioxide *carbondioxide = [CarbonDioxide
FindCarbonDioxide:@"YCO2MK01-123456.carbonDioxide"];
YCarbonDioxide *carbondioxide = [CarbonDioxide
FindCarbonDioxide:@"YCO2MK01-123456.MyFunction"];
YCarbonDioxide *carbondioxide = [CarbonDioxide FindCarbonDioxide:@"MyModule.carbonDioxide"
];
YCarbonDioxide *carbondioxide = [CarbonDioxide FindCarbonDioxide:@"MyModule.MyFunction"];

```



```
YCarbonDioxide *carbondioxide = [CarbonDioxide FindCarbonDioxide:@"MyFunction"];
```

[CarbonDioxide FindCarbonDioxide] returns an object which you can then use at will to control the CO2 sensor.

isOnline

The isOnline method of the object returned by [CarbonDioxide FindCarbonDioxide] 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 YCarbonDioxide.FindCarbonDioxide provides the CO2 level currently measured by the sensor. The value returned is a floating number in parts per million (volumic)

10.2. Control of the module part

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

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

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

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

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

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

Changing the module settings

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

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

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

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

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

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

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

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

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

Listing the modules

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

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

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

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

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

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

10.3. Error handling

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

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

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

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

As this latest situation is not the most desirable, the Yoctopuce library offers another possibility for error handling, allowing you to create a robust program without needing to catch exceptions at every line of code. You simply need to call the `YAPI.DisableExceptions()` function to commute the library to a mode where exceptions for all the functions are systematically replaced by specific return values, which can be tested by the caller when necessary. For each function, the name of each return value in case of error is systematically documented in the library reference. The name always follows the same logic: a `get_state()` method returns a `Y_STATE_INVALID` value, a

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

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

11. Using Yocto-CO2 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.

11.1. 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.

11.2. Using the Yoctopuce API in a Visual Basic project

The Visual Basic.NET Yoctopuce library is composed of a DLL and of source files in Visual Basic. The DLL is not a .NET DLL, but a classic DLL, written in C, which manages the low level communications with the 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.

11.3. Control of the CarbonDioxide function

A few lines of code are enough to use a Yocto-CO2. Here is the skeleton of a Visual Basic code snippet to use the CarbonDioxide function.

```
[...]
Dim errmsg As String
Dim carbondioxide As YCarbonDioxide

REM Get access to your device, connected locally on USB for instance
yRegisterHub("usb", errmsg)
carbondioxide = yFindCarbonDioxide("YCO2MK01-123456.carbonDioxide")

REM Hot-plug is easy: just check that the device is online
If (carbondioxide.isOnline()) Then
    REM Use carbondioxide.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.

yFindCarbonDioxide

The `yFindCarbonDioxide` function allows you to find a CO2 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-CO2 module with serial number `YCO2MK01-123456` which you have named `"MyModule"`, and for which you have given the `carbonDioxide` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
carbondioxide = yFindCarbonDioxide("YCO2MK01-123456.carbonDioxide")
carbondioxide = yFindCarbonDioxide("YCO2MK01-123456.MyFunction")
carbondioxide = yFindCarbonDioxide("MyModule.carbonDioxide")
carbondioxide = yFindCarbonDioxide("MyModule.MyFunction")
carbondioxide = yFindCarbonDioxide("MyFunction")
```

`yFindCarbonDioxide` returns an object which you can then use at will to control the CO2 sensor.

isOnline

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

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

get_currentValue

The `get_currentValue()` method of the object returned by `yFindCarbonDioxide` provides the CO2 level currently measured by the sensor. The value returned is a floating number in parts per million (volumic).

A real example

Launch Microsoft VisualBasic and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-CO2** 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 Sub
End Sub

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

    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 If

    If target = "any" Then
        tsensor = yFirstCarbonDioxide()
        If tsensor Is Nothing Then
            Console.WriteLine("No module connected (check USB cable) ")
        End If
        Console.WriteLine("using " + tsensor.get_module().get_serialNumber())
    Else
        tsensor = yFindCarbonDioxide(target + ".carbonDioxide")
    End If

    While (True)
        If Not (tsensor.isOnline()) Then
            Console.WriteLine("Module not connected (check identification and USB cable)")
        End If

        Console.WriteLine("CO2: " + Str(tsensor.get_currentValue()) + " ppm")
        Console.WriteLine(" (press Ctrl-C to exit)")
        ySleep(1000, errmsg)
    End While
    yFreeAPI()

End Sub

End Module
```

11.4. Control of the module part

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

```
Imports System.IO
Imports System.Environment

Module Module1

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

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

    If (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
yFreeAPI()
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.WriteLine("Module: serial= " + m.get_serialNumber())
        Console.WriteLine(" / name= " + m.get_logicalName())
    Else
        Console.WriteLine("not connected (check identification and USB cable)")
    End If
    yFreeAPI()

End Sub

End Module

```

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

Listing the modules

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

```

Module Module1

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

        REM Setup the API to use local USB devices
        If 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
        yFreeAPI()
    End Sub

End Module

```

End Module

11.5. Error handling

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

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

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

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

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

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

12. Using Yocto-CO2 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.

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

12.1. Installation

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

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

The Visual C#.NET Yoctopuce library is composed of a DLL and of source files in Visual C#. The DLL is not a .NET DLL, but a classic DLL, written in C, which manages the low level communications with the 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.

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

² www.yoctopuce.com/EN/libraries.php

³ The sources of this DLL are available in the C++ API

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

Then add in the same manner the `yapi.dll` DLL, located in the `Sources/dll` 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 CarbonDioxide function

A few lines of code are enough to use a Yocto-CO2. Here is the skeleton of a C# code snippet to use the `CarbonDioxide` function.

```
[...]
string errmsg = "";
YCarbonDioxide carbondioxide;

// Get access to your device, connected locally on USB for instance
YAPI.RegisterHub("usb", errmsg);
carbondioxide = YCarbonDioxide.FindCarbonDioxide("YCO2MK01-123456.carbonDioxide");

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

YCarbonDioxide.FindCarbonDioxide

The `YCarbonDioxide.FindCarbonDioxide` function allows you to find a CO2 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-CO2 module with serial number `YCO2MK01-123456` which you have named `"MyModule"`, and for which you have given the `carbonDioxide` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
carbondioxide = YCarbonDioxide.FindCarbonDioxide("YCO2MK01-123456.carbonDioxide");
carbondioxide = YCarbonDioxide.FindCarbonDioxide("YCO2MK01-123456.MyFunction");
carbondioxide = YCarbonDioxide.FindCarbonDioxide("MyModule.carbonDioxide");
carbondioxide = YCarbonDioxide.FindCarbonDioxide("MyModule.MyFunction");
carbondioxide = YCarbonDioxide.FindCarbonDioxide("MyFunction");
```

`YCarbonDioxide.FindCarbonDioxide` returns an object which you can then use at will to control the CO2 sensor.

isOnline

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

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

get_currentValue

The `get_currentValue()` method of the object returned by `YCarbonDioxide.FindCarbonDioxide` provides the CO2 level currently measured by the sensor. The value returned is a floating number in parts per million (volumic)

A real example

Launch Microsoft Visual C# and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-CO2** 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 ");
            Environment.Exit(0);
        }

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

            YCarbonDioxide co2sensor;

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

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

            if (target == "ANY") {
                co2sensor = YCarbonDioxide.FirstCarbonDioxide();

                if (co2sensor == null) {
                    Console.WriteLine("No module connected (check USB cable) ");
                    Environment.Exit(0);
                }
                Console.WriteLine("using " + co2sensor.get_module().get_serialNumber());
            } else {
                co2sensor = YCarbonDioxide.FindCarbonDioxide(target + ".carbonDioxide");
            }

            if (!co2sensor.isOnline()) {
                Console.WriteLine("Module not connected");
                Console.WriteLine("check identification and USB cable");
                Environment.Exit(0);
            }
            while (co2sensor.isOnline()) {
                Console.WriteLine("CO2: " + co2sensor.get_currentValue().ToString() + " ppm");
                Console.WriteLine(" (press Ctrl-C to exit)");

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

12.4. Control of the module part

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

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

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

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

            if (YAPI.RegisterHub("usb", ref errormsg) != YAPI.SUCCESS) {
                Console.WriteLine("RegisterHub error: " + errormsg);
                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.WriteLine("beacon:        ");
                if (m.get_beacon() == YModule.BEACON_ON)
                    Console.WriteLine("ON");
                else
                    Console.WriteLine("OFF");
                Console.WriteLine("upTime:        " + (m.get_upTime() / 1000 ).ToString() + " sec");
                Console.WriteLine("USB current:   " + m.get_usbCurrent().ToString() + " mA");
                Console.WriteLine("Logs:\r\n" + m.get_lastLogs());
            } else {
                Console.WriteLine(args[0] + " not connected (check identification and USB cable)");
            }
            YAPI.FreeAPI();
        }
    }
}
```

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

Changing the module settings

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

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

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

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

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

            if (YAPI.RegisterHub("usb", ref errormsg) != YAPI.SUCCESS) {
                Console.WriteLine("RegisterHub error: " + errormsg);
                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.WriteLine("Module: serial= " + m.get_serialNumber());
                Console.WriteLine(" / name= " + m.get_logicalName());
            } else {
                Console.WriteLine("not connected (check identification and USB cable)");
            }
            YAPI.FreeAPI();
        }
    }
}
```

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

Listing the modules

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

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

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

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

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

12.5. Error handling

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

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

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

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

As this latest situation is not the most desirable, the Yoctopuce library offers another possibility for error handling, allowing you to create a robust program without needing to catch exceptions at every line of code. You simply need to call the `YAPI.DisableExceptions()` function to commute the library to a mode where exceptions for all the functions are systematically replaced by specific return

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

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

13. Using the Yocto-CO2 with Universal Windows Platform

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

The Yoctopuce UWP library allows you to use Yoctopuce modules in a universal Windows application and is written in C# in its entirety. You can add it to a Visual Studio 2017¹ project.

13.1. Blocking and asynchronous functions

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

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

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

Example:

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

    return result;
}

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

¹ <https://www.visualstudio.com/vs/cordova/vs/>

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

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

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

13.2. Installation

Download the Yoctopuce library for \$LANG\$ from the Yoctopuce web site². There is no installation software, simply copy the content of the zip file in a directory of your choice. You essentially need the content of the `Sources` directory. The other directories contain documentation and a few sample programs. Sample projects are Visual Studio 2017 projects. Visual Studio 2017 is available on the Microsoft web site³.

13.3. Using the Yoctopuce API in a Visual Studio project

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

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

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

The Package.appxmanifest file

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

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

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

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

For the Yocto-CO2, here is what you must add in the "Capabilities" node:

```
<DeviceCapability Name="humaninterfacedevice">
  <!-- Yocto-CO2 -->
  <Device Id="vidpid:24e0 0027">
    <Function Type="usage:ff00 0001" />
  </Device>
</DeviceCapability>
```

² www.yoctopuce.com/EN/libraries.php

³ <https://www.visualstudio.com/downloads/>

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

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

13.4. Control of the CarbonDioxide function

A few lines of code are enough to use a Yocto-CO2. Here is the skeleton of a C# code snippet to use the CarbonDioxide function.

```
[...]

await YAPI.RegisterHub("usb");
carbondioxide = YCarbonDioxide.FindCarbonDioxide("YCO2MK01-123456.carbonDioxide");

//To manage hot-plug, we check that the module is here
if (await carbondioxide.isOnline()) {
    //Use carbondioxide.get_currentValue()
    ...
}

[...]
```

Let us look at these lines in more details.

YAPI.RegisterHub

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

YCarbonDioxide.FindCarbonDioxide

The `YCarbonDioxide.FindCarbonDioxide` function allows you to find a CO2 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-CO2 module with serial number *YCO2MK01-123456* which you have named *"MyModule"*, and for which you have given the *carbonDioxide* function the name *"MyFunction"*. The following five calls are strictly equivalent, as long as *"MyFunction"* is defined only once.

```
carbondioxide = YCarbonDioxide.FindCarbonDioxide("YCO2MK01-123456.carbonDioxide");
carbondioxide = YCarbonDioxide.FindCarbonDioxide("YCO2MK01-123456.MaFonction");
carbondioxide = YCarbonDioxide.FindCarbonDioxide("MonModule.carbonDioxide");
carbondioxide = YCarbonDioxide.FindCarbonDioxide("MonModule.MaFonction");
carbondioxide = YCarbonDioxide.FindCarbonDioxide("MaFonction");
```

`YCarbonDioxide.FindCarbonDioxide` returns an object which you can then use at will to control the CO2 sensor.

isOnline

The `isOnline()` method of the object returned by `YCarbonDioxide.FindCarbonDioxide` 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 `YCarbonDioxide.FindCarbonDioxide` provides the CO2 level currently measured by the sensor. The value returned is a floating number in parts per million (volumic)

13.5. A real example

Launch Visual Studio and open the corresponding project provided in the directory **Examples/Doc-GettingStarted-Yocto-CO2** of the Yoctopuce library.

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

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

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

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

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

                YCarbonDioxide co2sensor;

                if (Target.ToLower() == "any") {
                    co2sensor = YCarbonDioxide.FirstCarbonDioxide();

                    if (co2sensor == null) {
                        WriteLine("No module connected (check USB cable) ");
                        return -1;
                    }
                    YModule ymod = await co2sensor.get_module();
                    WriteLine("using " + await ymod.get_serialNumber());
                } else {
                    co2sensor = YCarbonDioxide.FindCarbonDioxide(Target + ".carbonDioxide");
                }

                while (await co2sensor.isOnline()) {
                    WriteLine("CO2: " + await co2sensor.get_currentValue() + " ppm");
                    await YAPI.Sleep(1000);
                }

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

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

13.6. Control of the module part

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

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

```

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

namespace Demo
{
    public class Demo : DemoBase
    {

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

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

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

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

```

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

Changing the module settings

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

```

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

namespace Demo
{
    public class Demo : DemoBase
    {

```

```

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

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

        await YAPI.RegisterHub(HubURL);

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

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

```

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

Listing the modules

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

```

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

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

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

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

```



```

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

```

13.7. Error handling

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

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

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

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

Example:

```

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

```


14. Using Yocto-CO2 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 CarbonDioxide 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_carbonDioxide;
```

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

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

³ www.yoctopuce.com/EN/libraries.php

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

```

Procedure Usage();
var
  exe : string;

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

var
  sensor : TYCarbonDioxide;
  errmsg : string;
  done : boolean;

begin
  if (paramcount<1) then usage();

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

  if paramstr(1)='any' then
  begin
    sensor := yFirstCarbonDioxide();
    if sensor=nil then
    begin
      writeln('No module connected (check USB cable)');
      halt;
    end
  end
  else
    sensor:= YFindCarbonDioxide(paramstr(1)+'.carbonDioxide');

  done:= false;
  repeat
    if (sensor.isOnline()) then
    begin
      Write('CO2: '+FloatToStr(sensor.get_currentValue())+' ppm');
      Writeln(' (press Ctrl-C to exit)');
      Sleep(1000);
    end
    else
    begin
      Writeln('Module not connected (check identification and USB cable)');
      done := true;
    end;
  until done;
  yFreeAPI();
end.

```

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

yocto_api and yocto_carbondioxide

These two units provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api` must always be used, `yocto_carbondioxide` is necessary to manage modules containing a CO2 sensor, such as Yocto-CO2.

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.

yFindCarbonDioxide

The `yFindCarbonDioxide` function allows you to find a CO2 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-CO2 module with serial number `YCO2MK01-123456` which you have named `"MyModule"`, and for which you have given the `carbonDioxide` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
carbondioxide := yFindCarbonDioxide("YCO2MK01-123456.carbonDioxide");
carbondioxide := yFindCarbonDioxide("YCO2MK01-123456.MyFunction");
carbondioxide := yFindCarbonDioxide("MyModule.carbonDioxide");
carbondioxide := yFindCarbonDioxide("MyModule.MyFunction");
carbondioxide := yFindCarbonDioxide("MyFunction");
```

`yFindCarbonDioxide` returns an object which you can then use at will to control the CO2 sensor.

isOnline

The `isOnline()` method of the object returned by `yFindCarbonDioxide` 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 `yFindCarbonDioxide` provides the CO2 level currently measured by the sensor. The value returned is a floating number in parts per million (volumic).

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 = 'YCO2MK01-123456'; // use serial number or logical name

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

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

```

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

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

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

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

```

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

Changing the module settings

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

```

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

const
  serial = 'YCO2MK01-123456'; // use serial number or logical name

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

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

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

  Writeln('Current logical name : '+module.get_logicalName());
  Write('Enter new name : ');
  Readln(newname);
  if (not(yCheckLogicalName(newname))) then

```

```

begin
    Writeln('invalid logical name');
    exit;
end;
module.set_logicalName(newname);
module.saveToFlash();
yFreeAPI();
Writeln('logical name is now : '+module.get_logicalName());
end.

```

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

Listing the modules

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

```

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

var
    module : TYModule;
    errmsg : string;

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

    Writeln('Device list');

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

```

14.4. Error handling

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

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

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

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

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

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

15. Using the Yocto-CO2 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 CarbonDioxide function

A few lines of code are enough to use a Yocto-CO2. Here is the skeleton of a Python code snippet to use the CarbonDioxide 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)
carbondioxide = YCarbonDioxide.FindCarbonDioxide("YCO2MK01-123456.carbonDioxide")

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

YCarbonDioxide.FindCarbonDioxide

The `YCarbonDioxide.FindCarbonDioxide` function allows you to find a CO2 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-CO2 module with serial number *YCO2MK01-123456* which you have named "MyModule", and for which you have given the *carbonDioxide* function the name "MyFunction". The following five calls are strictly equivalent, as long as "MyFunction" is defined only once.

```
carbondioxide = YCarbonDioxide.FindCarbonDioxide("YCO2MK01-123456.carbonDioxide")
carbondioxide = YCarbonDioxide.FindCarbonDioxide("YCO2MK01-123456.MyFunction")
carbondioxide = YCarbonDioxide.FindCarbonDioxide("MyModule.carbonDioxide")
carbondioxide = YCarbonDioxide.FindCarbonDioxide("MyModule.MyFunction")
carbondioxide = YCarbonDioxide.FindCarbonDioxide("MyFunction")
```

`YCarbonDioxide.FindCarbonDioxide` returns an object which you can then use at will to control the CO2 sensor.

isOnline

The `isOnline()` method of the object returned by `YCarbonDioxide.FindCarbonDioxide` 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 `YCarbonDioxide.FindCarbonDioxide` provides the CO2 level currently measured by the sensor. The value returned is a floating number in parts per million (volumic)

A real example

Launch Python and open the corresponding sample script provided in the directory **Examples/Doc-GettingStarted-Yocto-CO2** 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_carbondioxide 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)')

errmsg = YRefParam()

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

target = sys.argv[1]

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

if target == 'any':
    # retrieve any carbonDioxide sensor
    sensor = YCarbonDioxide.FirstCarbonDioxide()
    if sensor is None:
        die('No module connected')
    print("Using: " + sensor.get_module().get_serialNumber())
else:
    sensor = YCarbonDioxide.FindCarbonDioxide(target + '.carbonDioxide')

if not (sensor.isOnline()):
    die('device not connected')

while sensor.isOnline():
    print("CO2 : " + "%2.1f" % sensor.get_currentValue() + "ppm (Ctrl-C to stop)")
    YAPI.Sleep(1000)

YAPI.FreeAPI()

```

15.4. Control of the module part

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

```

#!/usr/bin/python
# -*- coding: utf-8 -*-
import os, sys

from yocto_api import *

def usage():
    sys.exit("usage: demo <serial or logical name> [ON/OFF]")

errmsg = YRefParam()
if YAPI.RegisterHub("usb", errmsg) != YAPI.SUCCESS:
    sys.exit("RegisterHub error: " + str(errmsg))

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

m = YModule.FindModule(sys.argv[1]) # use serial or logical name

if m.isOnline():
    if len(sys.argv) > 2:
        if sys.argv[2].upper() == "ON":
            m.set_beacon(YModule.BEACON_ON)
        if sys.argv[2].upper() == "OFF":

```

```

        m.set_beacon(YModule.BEACON_OFF)

    print("serial:      " + m.get_serialNumber())
    print("logical name: " + m.get_logicalName())
    print("luminosity:   " + str(m.get_luminosity()))
    if m.get_beacon() == YModule.BEACON_ON:
        print("beacon:      ON")
    else:
        print("beacon:      OFF")
    print("upTime:        " + str(m.get_upTime() / 1000) + " sec")
    print("USB current:   " + str(m.get_usbCurrent()) + " mA")
    print("logs:\n" + m.get_lastLogs())
else:
    print(sys.argv[1] + " not connected (check identification and USB cable)")
YAPI.FreeAPI()

```

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

Changing the module settings

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

```

#!/usr/bin/python
# -*- coding: utf-8 -*-
import os, sys

from yocto_api import *

def usage():
    sys.exit("usage: demo <serial or logical name> <new logical name>")

if len(sys.argv) != 3:
    usage()

errmsg = YRefParam()
if YAPI.RegisterHub("usb", errmsg) != YAPI.SUCCESS:
    sys.exit("RegisterHub error: " + str(errmsg))

m = YModule.FindModule(sys.argv[1]) # use serial or logical name
if m.isOnline():
    newname = sys.argv[2]
    if not YAPI.CheckLogicalName(newname):
        sys.exit("Invalid name (" + newname + ")")
    m.set_logicalName(newname)
    m.saveToFlash() # do not forget this
    print("Module: serial= " + m.get_serialNumber() + " / name= " + m.get_logicalName())
else:
    sys.exit("not connected (check identification and USB cable)")
YAPI.FreeAPI()

```

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

Listing the modules

Obtaining the list of the connected modules is performed with the `YModule.yFirstModule()` function which returns the first module found. Then, you only need to call the `nextModule()`

function of this object to find the following modules, and this as long as the returned value is not null. Below a short example listing the connected modules.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import os, sys

from yocto_api import *

errmsg = YRefParam()

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

print('Device list')

module = YModule.FirstModule()
while module is not None:
    print(module.get_serialNumber() + ' (' + module.get_productName() + ')')
    module = module.nextModule()
YAPI.FreeAPI()
```

15.5. Error handling

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

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

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

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

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

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

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

16. Using the Yocto-CO2 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 CarbonDioxide function

A few lines of code are enough to use a Yocto-CO2. Here is the skeleton of a Java code snippet to use the CarbonDioxide function.

```
[...]  
  
// Get access to your device, connected locally on USB for instance  
YAPI.RegisterHub("127.0.0.1");  
carbondioxide = YCarbonDioxide.FindCarbonDioxide("YCO2MK01-123456.carbonDioxide");  
  
// Hot-plug is easy: just check that the device is online  
if (carbondioxide.isOnline())  
{  
    // Use carbondioxide.get_currentValue()  
    [...]  
}
```

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.

YCarbonDioxide.FindCarbonDioxide

The `YCarbonDioxide.FindCarbonDioxide` function allows you to find a CO2 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-CO2 module with serial number *YCO2MK01-123456* which you have named *"MyModule"*, and for which you have given the *carbonDioxide* function the name *"MyFunction"*. The following five calls are strictly equivalent, as long as *"MyFunction"* is defined only once.

```
carbondioxide = YCarbonDioxide.FindCarbonDioxide("YCO2MK01-123456.carbonDioxide")
carbondioxide = YCarbonDioxide.FindCarbonDioxide("YCO2MK01-123456.MyFunction")
carbondioxide = YCarbonDioxide.FindCarbonDioxide("MyModule.carbonDioxide")
carbondioxide = YCarbonDioxide.FindCarbonDioxide("MyModule.MyFunction")
carbondioxide = YCarbonDioxide.FindCarbonDioxide("MyFunction")
```

`YCarbonDioxide.FindCarbonDioxide` returns an object which you can then use at will to control the CO2 sensor.

isOnline

The `isOnline()` method of the object returned by `YCarbonDioxide.FindCarbonDioxide` 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 `YCarbonDioxide.FindCarbonDioxide` provides the CO2 level currently measured by the sensor. The value returned is a floating number in parts per million (volumic)

A real example

Launch your Java environment and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-CO2** 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);
        }
        YCarbonDioxide co2sensor;
        if (args.length > 0) {
            co2sensor = YCarbonDioxide.FindCarbonDioxide(args[0] + ".carbonDioxide");
        } else {
            co2sensor = YCarbonDioxide.FirstCarbonDioxide();
            if (co2sensor == null) {
```



```

        System.out.println("No module connected (check USB cable)");
        System.exit(1);
    }
}
while (true) {
    try {
        System.out.println("CO2: "+co2sensor.getCurrentValue() + " ppm");
        System.out.println("  (press Ctrl-C to exit)");
        YAPI.Sleep(1000);
    } catch (YAPI_Exception ex) {
        System.out.println("Module not connected (check identification and USB
cable)");
        break;
    }
};
YAPI.FreeAPI();
}
}

```

16.3. Control of the module part

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

```

import com.yoctopuce.YoctoAPI.*;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Demo {

    public static void main(String[] args)
    {
        try {
            // setup the API to use local VirtualHub
            YAPI.RegisterHub("127.0.0.1");
        } catch (YAPI_Exception ex) {
            System.out.println("Cannot contact VirtualHub on 127.0.0.1 (" +
ex.getLocalizedMessage() + ")");
            System.out.println("Ensure that the VirtualHub application is running");
            System.exit(1);
        }
        System.out.println("usage: demo [serial or logical name] [ON/OFF]");

        YModule module;
        if (args.length == 0) {
            module = YModule.FirstModule();
            if (module == null) {
                System.out.println("No module connected (check USB cable)");
                System.exit(1);
            }
        } else {
            module = YModule.FindModule(args[0]); // use serial or logical name
        }

        try {
            if (args.length > 1) {
                if (args[1].equalsIgnoreCase("ON")) {
                    module.setBeacon(YModule.BEACON_ON);
                } else {
                    module.setBeacon(YModule.BEACON_OFF);
                }
            }
            System.out.println("serial:      " + module.get_serialNumber());
            System.out.println("logical name: " + module.get_logicalName());
            System.out.println("luminosity:  " + module.get_luminosity());
            if (module.get_beacon() == YModule.BEACON_ON) {
                System.out.println("beacon:      ON");
            } else {
                System.out.println("beacon:      OFF");
            }
            System.out.println("upTime:      " + module.get_upTime() / 1000 + " sec");
            System.out.println("USB current: " + module.get_usbCurrent() + " mA");
        }
    }
}

```

```

        System.out.println("logs:\n" + module.get_lastLogs());
    } catch (YAPI_Exception ex) {
        System.out.println(args[1] + " not connected (check identification and USB
cable)");
    }
    YAPI.FreeAPI();
}
}

```

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

Changing the module settings

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

```

import com.yoctopuce.YoctoAPI.*;

public class Demo {

    public static void main(String[] args)
    {
        try {
            // setup the API to use local VirtualHub
            YAPI.RegisterHub("127.0.0.1");
        } catch (YAPI_Exception ex) {
            System.out.println("Cannot contact VirtualHub on 127.0.0.1 (" +
ex.getLocalizedMessage() + ")");
            System.out.println("Ensure that the VirtualHub application is running");
            System.exit(1);
        }

        if (args.length != 2) {
            System.out.println("usage: demo <serial or logical name> <new logical name>");
            System.exit(1);
        }

        YModule m;
        String newname;

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

        try {
            newname = args[1];
            if (!YAPI.CheckLogicalName(newname))
            {
                System.out.println("Invalid name (" + newname + ")");
                System.exit(1);
            }

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

            System.out.println("Module: serial= " + m.get_serialNumber());
            System.out.println(" / name= " + m.get_logicalName());
        } catch (YAPI_Exception ex) {
            System.out.println("Module " + args[0] + "not connected (check identification
and USB cable)");
            System.out.println(ex.getMessage());
            System.exit(1);
        }

        YAPI.FreeAPI();
    }
}

```

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

Listing the modules

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

```
import com.yoctopuce.YoctoAPI.*;

public class Demo {

    public static void main(String[] args)
    {
        try {
            // setup the API to use local VirtualHub
            YAPI.RegisterHub("127.0.0.1");
        } catch (YAPI_Exception ex) {
            System.out.println("Cannot contact VirtualHub on 127.0.0.1 (" +
ex.getLocalizedMessage() + ")");
            System.out.println("Ensure that the VirtualHub application is running");
            System.exit(1);
        }

        System.out.println("Device list");
        YModule module = YModule.FirstModule();
        while (module != null) {
            try {
                System.out.println(module.get_serialNumber() + " (" +
module.get_productName() + ")");
            } catch (YAPI_Exception ex) {
                break;
            }
            module = module.nextModule();
        }
        YAPI.FreeAPI();
    }
}
```

16.4. Error handling

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

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

In the Java API, error handling is implemented with exceptions. Therefore you must catch and handle correctly all exceptions that might be thrown by the API if you do not want your software to crash as soon as you unplug a device.

17. Using the Yocto-CO2 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 Tranformer Pad TF300T
- Kurio 7

If your Android machine is not able to control Yoctopuce modules natively, you still have the possibility to remotely control modules driven by a VirtualHub on another OS, or a YoctoHub ².

17.4. Activating the USB port under Android

By default, Android does not allow an application to access the devices connected to the USB port. To enable your application to interact with a Yoctopuce module directly connected on your tablet on a USB port, a few additional steps are required. If you intend to interact only with modules connected on another machine through the network, you can ignore this section.

In your `AndroidManifest.xml`, you must declare using the "USB Host" functionality by adding the `<uses-feature android:name="android.hardware.usb.host" />` tag in the `manifest` section.

```
<manifest ...>
...
<uses-feature android:name="android.hardware.usb.host" />;
...
</manifest>
```

When first accessing a Yoctopuce module, Android opens a window to inform the user that the application is going to access the connected module. The user can deny or authorize access to the device. If the user authorizes the access, the application can access the connected device as long as

² Yoctohubs are a plug and play way to add network connectivity to your Yoctopuce devices. more info on <http://www.yoctopuce.com/EN/products/category/extensions-and-networking>

it stays connected. To enable the Yoctopuce library to correctly manage these authorizations, 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 CarbonDioxide function

A few lines of code are enough to use a Yocto-CO2. Here is the skeleton of a Java code snippet to use the CarbonDioxide function.

```
[...]

// Retrieving the object representing the module (connected here locally by USB)
YAPI.EnableUSBHost(this);
YAPI.RegisterHub("usb");
carbondioxide = YCarbonDioxide.FindCarbonDioxide("YCO2MK01-123456.carbonDioxide");

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

YCarbonDioxide.FindCarbonDioxide

The `YCarbonDioxide.FindCarbonDioxide` function allows you to find a CO2 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-CO2 module with serial number *YCO2MK01-123456* which you have named *"MyModule"*, and for which you have given the *carbonDioxide* function the name *"MyFunction"*. The following five calls are strictly equivalent, as long as *"MyFunction"* is defined only once.

```
carbondioxide = YCarbonDioxide.FindCarbonDioxide("YCO2MK01-123456.carbonDioxide")
carbondioxide = YCarbonDioxide.FindCarbonDioxide("YCO2MK01-123456.MyFunction")
carbondioxide = YCarbonDioxide.FindCarbonDioxide("MyModule.carbonDioxide")
carbondioxide = YCarbonDioxide.FindCarbonDioxide("MyModule.MyFunction")
carbondioxide = YCarbonDioxide.FindCarbonDioxide("MyFunction")
```

`YCarbonDioxide.FindCarbonDioxide` returns an object which you can then use at will to control the CO2 sensor.

isOnline

The `isOnline()` method of the object returned by `YCarbonDioxide.FindCarbonDioxide` 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 `YCarbonDioxide.FindCarbonDioxide` provides the CO2 level currently measured by the sensor. The value returned is a floating number in parts per million (volumic)

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.OnItemClickListener;
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.YCarbonDioxide;
import com.yoctopuce.YoctoAPI.YModule;

public class GettingStarted_Yocto_CO2 extends Activity implements OnItemClickListener
{
    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_co2);
        Spinner my_spin = (Spinner) findViewById(R.id.spinner1);
        my_spin.setOnItemClickListener(this);
        aa = new ArrayAdapter<String>(this, android.R.layout.simple_spinner_item);
        aa.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
        my_spin.setAdapter(aa);
        handler = new Handler();
    }

    @Override
    protected void onStart()
    {
        super.onStart();
        try {
            aa.clear();
            YAPI.EnableUSBHost(this);
            YAPI.RegisterHub("usb");
            YModule module = YModule.FirstModule();
            while (module != null) {
                if (module.get_productName().equals("Yocto-CO2")) {
                    String serial = module.get_serialNumber();
                    aa.add(serial);
                }
                module = module.nextModule();
            }
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
        aa.notifyDataSetChanged();
        handler.postDelayed(r, 500);
    }

    @Override
    protected void onStop()
    {
        super.onStop();
        handler.removeCallbacks(r);
        YAPI.FreeAPI();
    }
}
```

```

@Override
public void onItemClick(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) {
            YCarbonDioxide co2_sensor = YCarbonDioxide.FindCarbonDioxide(serial +
".carbonDioxide");
            try {
                TextView view = (TextView) findViewById(R.id.co2field);
                view.setText(String.format("%.1f %s", co2_sensor.getCurrentValue(),
co2_sensor.getUnit()));
            } catch (YAPI_Exception e) {
                e.printStackTrace();
            }
            handler.postDelayed(this, 1000);
        }
    };
}

```

17.6. Control of the module part

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

```

package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
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 OnItemClickListener
{
    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.setOnItemClickListener(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.getLogiicalName());
        field = (TextView) findViewById(R.id.luminosityfield);
        field.setText(String.format("%d%%", module.getLuminosity()));
        field = (TextView) findViewById(R.id.uptimefield);
        field.setText(module.getUpTime() / 1000 + " sec");
        field = (TextView) findViewById(R.id.usbcurrentfield);
        field.setText(module.getUsbCurrent() + " mA");
        Switch sw = (Switch) findViewById(R.id.beaconswitch);
        sw.setChecked(module.getBeacon() == YModule.BEACON_ON);
        field = (TextView) findViewById(R.id.logs);
        field.setText(module.get_lastLogs());

    } catch (YAPI_Exception e) {
        e.printStackTrace();
    }
}

@Override
public void onItemSelected(AdapterView<?> parent, View view, int pos, long id)
{
    String hwid = parent.getItemAtPosition(pos).toString();
    module = YModule.FindModule(hwid);
    DisplayModuleInfo();
}

@Override
public void onNothingSelected(AdapterView<?> arg0)
{
}

public void refreshInfo(View view)
{
    DisplayModuleInfo();
}

public void toggleBeacon(View view)
{
    if (module == null)
        return;
    boolean on = ((Switch) view).isChecked();

    try {
        if (on) {

```

```

        module.setBeacon(YModule.BEACON_ON);
    } else {
        module.setBeacon(YModule.BEACON_OFF);
    }
} catch (YAPI_Exception e) {
    e.printStackTrace();
}
}
}

```

Each property `xxx` of the module can be read thanks to a method of type `YModule.get_xxx()`, 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.OnItemClickListener;
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 OnItemClickListener
{
    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.setOnItemClickListener(this);
        aa = new ArrayAdapter<String>(this, android.R.layout.simple_spinner_item);
        aa.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
        my_spin.setAdapter(aa);
    }

    @Override
    protected void onStart()
    {
        super.onStart();

        try {
            aa.clear();
            YAPI.EnableUSBHost(this);
            YAPI.RegisterHub("usb");
            YModule r = YModule.FirstModule();
            while (r != null) {
                String hwid = r.get_hardwareId();
                aa.add(hwid);
                r = r.nextModule();
            }
        }
    }
}

```

```

        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
        // refresh Spinner with detected relay
        aa.notifyDataSetChanged();
    }

    @Override
    protected void onStop()
    {
        super.onStop();
        YAPI.FreeAPI();
    }

    private void DisplayModuleInfo()
    {
        TextView field;
        if (module == null)
            return;
        try {
            YAPI.UpdateDeviceList(); // fixme
            field = (TextView) findViewById(R.id.logicalnamefield);
            field.setText(module.getLogicalName());
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
    }

    @Override
    public void onItemClick(AdapterView<?> parent, View view, int pos, long id)
    {
        String hwid = parent.getItemAtPosition(pos).toString();
        module = YModule.FindModule(hwid);
        DisplayModuleInfo();
    }

    @Override
    public void onNothingSelected(AdapterView<?> arg0)
    {
    }

    public void saveName(View view)
    {
        if (module == null)
            return;

        EditText edit = (EditText) findViewById(R.id.newname);
        String newname = edit.getText().toString();
        try {
            if (!YAPI.CheckLogicalName(newname)) {
                Toast.makeText(getApplicationContext(), "Invalid name (" + newname + ")",
                    Toast.LENGTH_LONG).show();
                return;
            }
            module.set_logicalName(newname);
            module.saveToFlash(); // do not forget this
            edit.setText("");
        } catch (YAPI_Exception ex) {
            ex.printStackTrace();
        }
        DisplayModuleInfo();
    }
}

```

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

Listing the modules

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

```
package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.util.TypedValue;
import android.view.View;
import android.widget.LinearLayout;
import android.widget.TextView;

import com.yoctopuce.YoctoAPI.YAPI;
import com.yoctopuce.YoctoAPI.YAPI_Exception;
import com.yoctopuce.YoctoAPI.YModule;

public class Inventory extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.inventory);
    }

    public void refreshInventory(View view)
    {
        LinearLayout layout = (LinearLayout) findViewById(R.id.inventoryList);
        layout.removeAllViews();

        try {
            YAPI.UpdateDeviceList();
            YModule module = YModule.FirstModule();
            while (module != null) {
                String line = module.get_serialNumber() + " (" + module.get_productName() +
                ")";

                TextView tx = new TextView(this);
                tx.setText(line);
                tx.setTextSize(TypedValue.COMPLEX_UNIT_SP, 20);
                layout.addView(tx);
                module = module.nextModule();
            }
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
    }

    @Override
    protected void onStart()
    {
        super.onStart();
        try {
            YAPI.EnableUSBHost(this);
            YAPI.RegisterHub("usb");
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
        refreshInventory(null);
    }

    @Override
    protected void onStop()
    {
        super.onStop();
        YAPI.FreeAPI();
    }
}
```

17.7. Error handling

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

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

In the Java API for Android, error handling is implemented with exceptions. Therefore you must catch and handle correctly all exceptions that might be thrown by the API if you do not want your software to crash soon as you unplug a device.

18. Advanced programming

The preceding chapters have introduced, in each available language, the basic programming functions which can be used with your Yocto-CO2 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.

18.1. Event programming

The methods to manage Yoctopuce modules which we presented to you in preceding chapters were polling functions, consisting in permanently asking the API if something had changed. While easy to understand, this programming technique is not the most efficient, nor the most reactive. Therefore, the Yoctopuce programming API also provides an event programming model. This technique consists in asking the API to signal by itself the important changes as soon as they are detected. Each time a key parameter is modified, the API calls a callback function which you have defined in advance.

Detecting module arrival and departure

Hot-plug management is important when you work with USB modules because, sooner or later, you will have to connect or disconnect a module when your application is running. The API is designed to manage module unexpected arrival or departure in a transparent way. But your application must take this into account if it wants to avoid pretending to use a disconnected module.

Event programming is particularly useful to detect module connection/disconnection. Indeed, it is simpler to be told of new connections rather than to have to permanently list the connected modules to deduce which ones just arrived and which ones left. To be warned as soon as a module is connected, you need three pieces of code.

The callback

The callback is the function which is called each time a new Yoctopuce module is connected. It takes as parameter the relevant module.

```
static void deviceArrival(YModule m)
{
    Console.WriteLine("New module : " + m.get_serialNumber());
}
```

Initialization

You must then tell the API that it must call the callback when a new module is connected.

```
YAPI.RegisterDeviceArrivalCallback(deviceArrival);
```

Note that if modules are already connected when the callback is registered, the callback is called for each of the already connected modules.

Triggering callbacks

A classis issue of callback programming is that these callbacks can be triggered at any time, including at times when the main program is not ready to receive them. This can have undesired side effects, such as dead-locks and other race conditions. Therefore, in the Yoctopuce API, module arrival/departure callbacks are called only when the `UpdateDeviceList()` function is running. You only need to call `UpdateDeviceList()` at regular intervals from a timer or from a specific thread to precisely control when the calls to these callbacks happen:

```
// waiting loop managing callbacks
while (true)
{
    // module arrival / departure callback
    YAPI.UpdateDeviceList(ref errmsg);
    // non active waiting time managing other callbacks
    YAPI.Sleep(500, ref errmsg);
}
```

In a similar way, it is possible to have a callback when a module is disconnected. You can find a complete example implemented in your favorite programming language in the *Examples/Prog-EventBased* directory of the corresponding library.

Be aware that in most programming languages, callbacks must be global procedures, and not methods. If you wish for the callback to call the method of an object, define your callback as a global procedure which then calls your method.

Detecting a modification in the value of a sensor

The Yoctopuce API also provides a callback system allowing you to be notified automatically with the value of any sensor, either when the value has changed in a significant way or periodically at a preset frequency. The code necessary to do so is rather similar to the code used to detect when a new module has been connected.

This technique is useful in particular if you want to detect very quick value changes (within a few milliseconds), as it is much more efficient than reading repeatedly the sensor value and therefore gives better performances.

Callback invocation

To enable a better control, value change callbacks are only called when the `YAPI.Sleep()` and `YAPI.HandleEvents()` functions are running. Therefore, you must call one of these functions at a regular interval, either from a timer or from a parallel thread.

```
while (true)
{
    // inactive waiting loop allowing you to trigger
    // value change callbacks
    YAPI.Sleep(500, ref errmsg);
}
```

In programming environments where only the interface thread is allowed to interact with the user, it is often appropriate to call `YAPI.HandleEvents()` from this thread.

The value change callback

This type of callback is called when a CO2 sensor changes in a significant way. It takes as parameter the relevant function and the new value, as a character string.¹

```
static void valueChangeCallback(YCarbonDioxide fct, string value)
{
    Console.WriteLine(fct.get_hardwareId() + "=" + value);
}
```

In most programming languages, callbacks are global procedures, not methods. If you wish for the callback to call a method of an object, define your callback as a global procedure which then calls your method. If you need to keep a reference to your object, you can store it directly in the YCarbonDioxide 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 CarbonDioxide function with the help of the `registerValueCallback` method. The following example sets up a callback for the first available CarbonDioxide function.

```
YCarbonDioxide f = YCarbonDioxide.FirstCarbonDioxide();
f.registerValueCallback(carbonDioxideChangeCallBack)
```

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(YCarbonDioxide fct, YMeasure measure)
{
    Console.WriteLine(fct.get_hardwareId() + "=" +
        measure.get_averageValue());
}
```

Setting up a timed report callback

The callback is set up for a given CarbonDioxide function with the help of the `registerTimedReportCallback` method. The callback will only be invoked once a callback frequency as been set using `set_reportFrequency` (which defaults to timed report callback turned off). The frequency is specified as a string (same as for the data logger), by specifying the number of calls per second (/s), per minute (/m) or per hour (/h). The maximal frequency is 100 times per second (i.e. "100/s"), and the minimal frequency is 1 time per hour (i.e. "1/h"). When the frequency is higher than or equal to 1/s, the measure represents an instant value. When the frequency is below, the measure will include distinct minimal, maximal and average values based on a sampling performed automatically by the device.

The following example sets up a timed report callback 4 times per minute for the first available CarbonDioxide function.

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

```
YCarbonDioxide f = YCarbonDioxide.FirstCarbonDioxide();
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 `YCarbonDioxide`. Thus, the same callback function will be usable with any subclass of `YSensor` (and in particular with `YCarbonDioxide`). 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.

18.2. The data logger

Your Yocto-CO2 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-CO2 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-CO2 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-CO2 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-CO2 does not have a battery, it cannot guess alone the current time when powered on. Nevertheless, the Yocto-CO2 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-CO2 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-CO2 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-CO2 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-CO2 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-CO2 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-CO2 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-CO2 might be subject to a clock skew (theoretically up to 0.3%).

18.3. Sensor calibration

Your Yocto-CO2 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-CO2. 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-CO2 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-CO2. On the other hand, the API supports an external arbitrary calibration, implemented with callbacks.

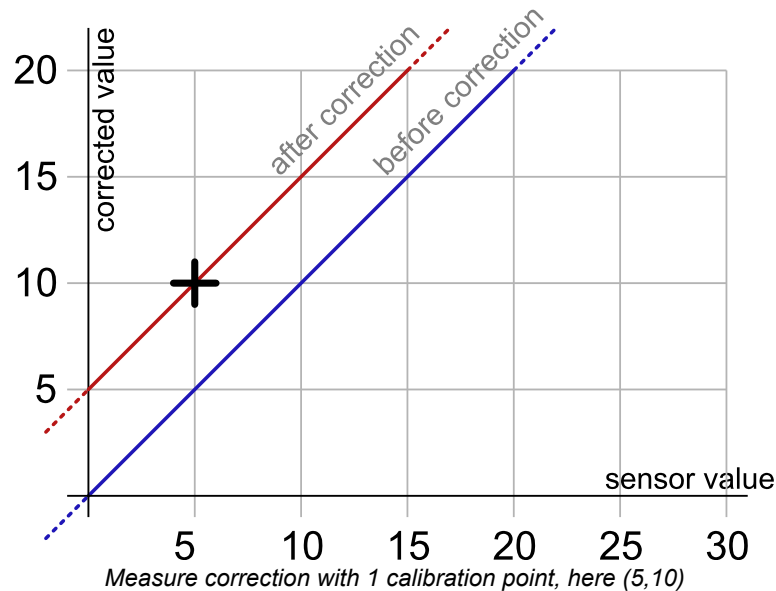
1 to 5 point linear interpolation

These transformations are performed directly inside the Yocto-CO2 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 `carbonDioxide` function returns *b*.

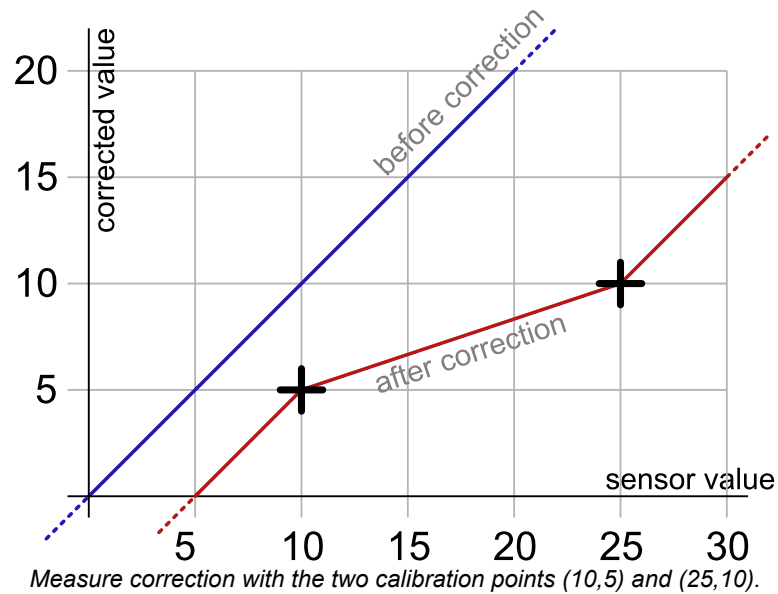


The application is very simple: you only need to call the `calibrateFromPoints()` method of the function you wish to correct. The following code applies the correction illustrated on the graph above to the first `carbonDioxide` function found. Note the call to the `saveToFlash` method of the module hosting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

```
Double[] ValuesBefore = {5};
Double[] ValuesAfter = {10};
YCarbonDioxide f = YCarbonDioxide.FirstCarbonDioxide();
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 `carbonDioxide` 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.



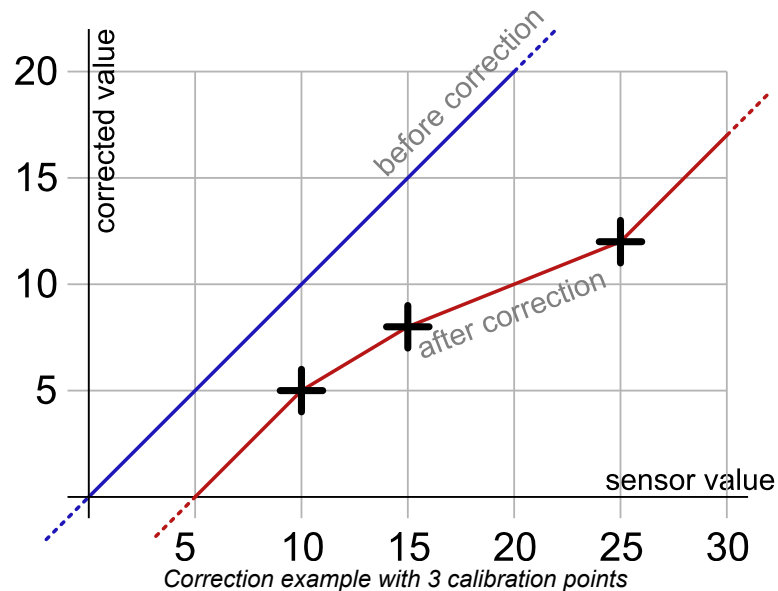
The code allowing you to program this calibration is very similar to the preceding code example.

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



```
f.calibrateFromPoints(ValuesBefore, ValuesAfter);
f.get_module().saveToFlash();
```

You will find, in the *Examples\Prog-Calibration* directory of the Delphi, VB, and C# libraries, an application allowing you to test the effects of the 1 to 5 point calibration.

Limitations

Due to storage and processing limitations of real values within Yoctopuce sensors, raw values and corrected values must conform to a few numeric constraints:

- Only 3 decimals are taken into account (i.e. resolution is 0.001)
- The lowest allowed value is -2'100'000
- The highest allowed value is +2'100'000

Arbitrary interpolation

It is also possible to compute the interpolation instead of letting the module do it, in order to calculate a spline interpolation, for instance. To do so, you only need to store a callback in the API. This callback must specify the number of calibration points it is expecting.

```
public static double CustomInterpolation3Points(double rawValue, int calibType,
    int[] parameters, double[] beforeValues, double[] afterValues)
{
    double result;
    // the value to be corrected is rawValue
    // calibration points are in beforeValues and afterValues
    result = .... // interpolation of your choice
    return result;
}
YAPI.RegisterCalibrationHandler(3, CustomInterpolation3Points);
```

Note that these interpolation callbacks are global, and not specific to each function. Thus, each time someone requests a value from a module which contains in its flash memory the correct number of calibration points, the corresponding callback is called to correct the value before returning it, enabling thus a perfectly transparent measure correction.

19. Firmware Update

There are multiples way to update the firmware of a Yoctopuce module..

19.1. The VirtualHub or the YoctoHub

It is possible to update the firmware directly from the web interface of the VirtualHub or the YoctoHub. The configuration panel of the module has an "upgrade" button to start a wizard that will guide you through the firmware update procedure.

In case the firmware update fails for any reason, and the module does no start anymore, simply unplug the module then plug it back while maintaining the *Yocto-button* down. The module will boot in "firmware update" mode and will appear in the VirtualHub interface below the module list.

19.2. The command line library

All the command line tools can update Yoctopuce modules thanks to the `downloadAndUpdate` command. The module selection mechanism works like for a traditional command. The `[target]` is the name of the module that you want to update. You can also use the "any" or "all" aliases, or even a name list, where the names are separated by commas, without spaces.

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

The following example updates all the Yoctopuce modules connected by USB.

```
C:\>YModule all downloadAndUpdate
ok: Yocto-PowerRelay RELAYHI1-266C8(rev=15430) is up to date.
ok: 0 / 0 hubs in 0.000000s.
ok: 0 / 0 shields in 0.000000s.
ok: 1 / 1 devices in 0.130000s 0.130000s per device.
ok: All devices are now up to date.
C:\>
```

19.3. The Android application Yocto-Firmware

You can update your module firmware from your Android phone or tablet with the [Yocto-Firmware](#) application. This application lists all the Yoctopuce modules connected by USB and checks if a more recent firmware is available on www.yoctopuce.com. If a more recent firmware is available, you can

update the module. The application is responsible for downloading and installing the new firmware while preserving the module parameters.

Please note: while the firmware is being updated, the module restarts several times. Android interprets a USB device reboot as a disconnection and reconnection of the USB device and asks the authorization to use the USB port again. The user must click on *OK* for the update process to end successfully.

19.4. Updating the firmware with the programming library

If you need to integrate firmware updates in your application, the libraries offer you an API to update your modules.¹

Saving and restoring parameters

The `get_allSettings()` method returns a binary buffer enabling you to save a module persistent parameters. This function is very useful to save the network configuration of a YoctoHub for example.

```
YWireless wireless = YWireless.FindWireless("reference");
YModule m = wireless.get_module();
byte[] default_config = m.get_allSettings();
saveFile("default.bin", default_config);
...
```

You can then apply these parameters to other modules with the `set_allSettings()` method.

```
byte[] default_config = loadFile("default.bin");
YModule m = YModule.FirstModule();
while (m != null) {
    if (m.get_productName() == "YoctoHub-Wireless") {
        m.set_allSettings(default_config);
    }
    m = m.next();
}
```

Finding the correct firmware

The first step to update a Yoctopuce module is to find which firmware you must use. The `checkFirmware(path, onlynew)` method of the `YModule` object does exactly this. The method checks that the firmware given as argument (`path`) is compatible with the module. If the `onlynew` parameter is set, this method checks that the firmware is more recent than the version currently used by the module. When the file is not compatible (or if the file is older than the installed version), this method returns an empty string. In the opposite, if the file is valid, the method returns a file access path.

The following piece of code checks that the `c:\tmp\METEOMK1.17328.byn` is compatible with the module stored in the `m` variable .

```
YModule m = YModule.FirstModule();
...
...
string path = "c:\\tmp\\METEOMK1.17328.byn";
string newfirm = m.checkFirmware(path, false);
if (newfirm != "") {
    Console.WriteLine("firmware " + newfirm + " is compatible");
}
...
```

¹ The JavaScript, Node.js, and PHP libraries do not yet allow you to update the modules. These functions will be available in a next build.

The argument can be a directory (instead of a file). In this case, the method checks all the files of the directory recursively and returns the most recent compatible firmware. The following piece of code checks whether there is a more recent firmware in the `c:\tmp\` directory.

```
YModule m = YModule.FirstModule();
...
...
string path = "c:\\tmp";
string newfirm = m.checkFirmware(path, true);
if (newfirm != "") {
    Console.WriteLine("firmware " + newfirm + " is compatible and newer");
}
...
```

You can also give the `"www.yoctopuce.com"` string as argument to check whether there is a more recent published firmware on Yoctopuce's web site. In this case, the method returns the firmware URL. You can use this URL to download the firmware on your disk or use this URL when updating the firmware (see below). Obviously, this possibility works only if your machine is connected to Internet.

```
YModule m = YModule.FirstModule();
...
...
string url = m.checkFirmware("www.yoctopuce.com", true);
if (url != "") {
    Console.WriteLine("new firmware is available at " + url );
}
...
```

Updating the firmware

A firmware update can take several minutes. That is why the update process is run as a background task and is driven by the user code thanks to the `YFirmwareUpdate` class.

To update a Yoctopuce module, you must obtain an instance of the `YFirmwareUpdate` class with the `updateFirmware` method of a `YModule` object. The only parameter of this method is the *path* of the firmware that you want to install. This method does not immediately start the update, but returns a `YFirmwareUpdate` object configured to update the module.

```
string newfirm = m.checkFirmware("www.yoctopuce.com", true);
....
YFirmwareUpdate fw_update = m.updateFirmware(newfirm);
```

The `startUpdate()` method starts the update as a background task. This background task automatically takes care of

1. saving the module parameters
2. restarting the module in "update" mode
3. updating the firmware
4. starting the module with the new firmware version
5. restoring the parameters

The `get_progress()` and `get_progressMessage()` methods enable you to follow the progression of the update. `get_progress()` returns the progression as a percentage (100 = update complete). `get_progressMessage()` returns a character string describing the current operation (deleting, writing, rebooting, ...). If the `get_progress` method returns a negative value, the update process failed. In this case, the `get_progressMessage()` returns an error message.

The following piece of code starts the update and displays the progress on the standard output.

```
YFirmwareUpdate fw_update = m.updateFirmware(newfirm);
....
int status = fw_update.startUpdate();
while (status < 100 && status >= 0) {
```

```

int newstatus = fw_update.get_progress();
if (newstatus != status) {
    Console.WriteLine(status + "% "
        + fw_update.get_progressMessage());
}
YAPI.Sleep(500, ref errmsg);
status = newstatus;
}

if (status < 0) {
    Console.WriteLine("Firmware Update failed: "
        + fw_update.get_progressMessage());
} else {
    Console.WriteLine("Firmware Updated Successfully!");
}

```

An Android characteristic

You can update a module firmware using the Android library. However, for modules connected by USB, Android asks the user to authorize the application to access the USB port.

During firmware update, the module restarts several times. Android interprets a USB device reboot as a disconnection and a reconnection to the USB port, and prevents all USB access as long as the user has not closed the pop-up window. The user has to click on *OK* for the update process to continue correctly. **You cannot update a module connected by USB to an Android device without having the user interacting with the device.**

19.5. The "update" mode

If you want to erase all the parameters of a module or if your module does not start correctly anymore, you can install a firmware from the "update" mode.

To force the module to work in "update" mode, disconnect it, wait a few seconds, and reconnect it while maintaining the *Yocto-button* down. This will restart the module in "update" mode. This update mode is protected against corruptions and is always available.

In this mode, the module is not detected by the `YModule` objects anymore. To obtain the list of connected modules in "update" mode, you must use the `YAPI.GetAllBootLoaders()` function. This function returns a character string array with the serial numbers of the modules in "update" mode.

```
List<string> allBootLoader = YAPI.GetAllBootLoaders();
```

The update process is identical to the standard case (see the preceding section), but you must manually instantiate the `YFirmwareUpdate` object instead of calling `module.updateFirmware()`. The constructor takes as argument three parameters: the module serial number, the path of the firmware to be installed, and a byte array with the parameters to be restored at the end of the update (or null to restore default parameters).

```

YFirmwareUpdate fw_update;
fw_update = new YFirmwareUpdate(allBootLoader[0], newfirm, null);
int status = fw_update.startUpdate();
.....

```

20. 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.

20.1. 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.

20.2. 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-CO2 with the *YCO2MK01-12345* 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/YCO2MK01-12345/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/YCO2MK01-12345/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/YCO2MK01-12345/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 carbonDioxide function, build the following URL:

```
http://127.0.0.1:4444/bySerial/YCO2MK01-12345/api/carbonDioxide.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/YCO2MK01-12345/api/carbonDioxide/logicalName
```

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

```
http://127.0.0.1:4444/bySerial/YCO2MK01-12345/api/carbonDioxide?logicalName=myFunction
```

You can find the list of available attributes for your Yocto-CO2 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/YCO2MK01-12345/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/YCO2MK01-12345/dataLogger.json?
id=carbonDioxide&utc=1389801080
```

20.3. 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	Mac 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 buffsize, int *fullsize,
char *errmsg);
```

The *yapiInitAPI* function initializes the API and must be called once at the beginning of the program. For a USB type connection, the *connection_type* parameter takes value 1. The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message. This pointer can also point to *null*. The function returns a negative integer in case of error, zero otherwise.

The *yapiUpdateDeviceList* manages the inventory of connected Yoctopuce modules. It must be called at least once. To manage hot plug and detect potential newly connected modules, this function must be called at regular intervals. The *forceupdate* parameter must take value 1 to force a hardware scan. The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message. This pointer can also point to *null*. The function returns a negative integer in case of error, zero otherwise.

Finally, the *yapiHTTPRequest* function sends HTTP requests to the module REST API. The *device* parameter contains the serial number or the logical name of the module which you want to reach. The *request* parameter contains the full HTTP request (including terminal line breaks). *buffer* points to a character buffer long enough to contain the answer. *buffersize* 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;
                        buffsize:integer;var fullsize:integer;
                        errmsg : pansichar):integer;cdecl;
                        external 'yapi.dll' name 'yapiHTTPRequest';

var
    errmsgBuffer : array [0..256] of ansichar;
    dataBuffer   : array [0..1024] of ansichar;
    errmsg,data   : pansichar;
    fullsize,p    : integer;

const
    serial       = 'YCO2MK01-12345';
    getValue = 'GET /api/module/luminosity HTTP/1.1'#13#10#13#10;
    setValue = 'GET /api/module?luminosity=100 HTTP/1.1'#13#10#13#10;

begin
    errmsg := @errmsgBuffer;
    data := @dataBuffer;
    // API initialization
    if(yapiInitAPI(1,errmsg)<0) then
        begin
            writeln(errmsg);
            halt;
        end;

    // forces a device inventory
    if( yapiUpdateDeviceList(1,errmsg)<0) then
        begin
            writeln(errmsg);
            halt;
        end;

    // requests the module luminosity
    if (yapiHTTPRequest(serial,getValue,data,sizeof(dataBuffer),fullsize,errmsg)<0) then
        begin
            writeln(errmsg);
            halt;
        end;

    // searches for the HTTP header end
    p := pos(#13#10#13#10,data);

    // displays the response minus the HTTP header
    writeln(copy(data,p+4,length(data)-p-3));

    // changes the luminosity
    if (yapiHTTPRequest(serial,setValue,data,sizeof(dataBuffer),fullsize,errmsg)<0) then
        begin
            writeln(errmsg);
            halt;
        end;

end.

```

Module inventory

To perform an inventory of Yoctopuce modules, you need two functions from the dynamic library:

```

int yapiGetAllDevices(int *buffer,int maxsize,int *neededsize,char *errmsg);
int yapiGetDeviceInfo(int devdesc,yDeviceSt *infos, char *errmsg);

```

The *yapiGetAllDevices* function retrieves the list of all connected modules as a list of handles. *buffer* points to a 32-bit integer array which contains the returned handles. *maxsize* is the size in bytes of the buffer. To *neededsize* is assigned the necessary size to store all the handles. From this, you can deduce either the number of connected modules or that the input buffer is too small. The *errmsg*

parameter must point to a 255 character buffer to retrieve a potential error message. This pointer can also point to *null*. The function returns a negative integer in case of error, zero otherwise.

The *yapiGetDeviceInfo* function retrieves the information related to a module from its handle. *devdesc* is a 32-bit integer representing the module and which was obtained through *yapiGetAllDevices*. *infos* points to a data structure in which the result is stored. This data structure has the following format:

Name	Type	Size (bytes)	Description
vendorid	int	4	Yoctopuce USB ID
deviceid	int	4	Module USB ID
devrelease	int	4	Module version
nbinbterfaces	int	4	Number of USB interfaces used by the module
manufacturer	char[]	20	Yoctopuce (null terminated)
productname	char[]	28	Model (null terminated)
serial	char[]	20	Serial number (null terminated)
logicalname	char[]	20	Logical name (null terminated)
firmware	char[]	22	Firmware version (null terminated)
beacon	byte	1	Beacon state (0/1)

The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message.

Here is a sample program written in pascal using the *yapi.dll* DLL to list the connected modules.

```
// device description structure
type yDeviceSt = packed record
  vendorid      : word;
  deviceid      : word;
  devrelease    : word;
  nbinbterfaces : word;
  manufacturer  : array [0..19] of ansichar;
  productname   : array [0..27] of ansichar;
  serial        : array [0..19] of ansichar;
  logicalname    : array [0..19] of ansichar;
  firmware      : array [0..21] of ansichar;
  beacon        : byte;
end;

// Dll function import
function yapiInitAPI(mode:integer;
  errmsg : pansichar):integer;cdecl;
  external 'yapi.dll' name 'yapiInitAPI';

function yapiUpdateDeviceList(force:integer;errmsg : pansichar):integer;cdecl;
  external 'yapi.dll' name 'yapiUpdateDeviceList';

function yapiGetAllDevices( buffer:pointer;
  maxsize:integer;
  var neededsize:integer;
  errmsg : pansichar):integer; cdecl;
  external 'yapi.dll' name 'yapiGetAllDevices';

function apiGetDeviceInfo(d:integer; var infos:yDeviceSt;
  errmsg : pansichar):integer; cdecl;
  external 'yapi.dll' name 'yapiGetDeviceInfo';

var
  errmsgBuffer : array [0..256] of ansichar;
  dataBuffer    : array [0..127] of integer; // max of 128 USB devices
  errmsg,data   : pansichar;
  neededsize,i  : integer;
  devinfos      : yDeviceSt;

begin
  errmsg := @errmsgBuffer;

  // API initialization
  if(yapiInitAPI(1,errmsg)<0) then
    begin
      writeln(errmsg);
```

```

    halt;
end;

// forces a device inventory
if( yapiUpdateDeviceList(1,errmsg)<0) then
begin
    writeln(errmsg);
    halt;
end;

// loads all device handles into dataBuffer
if yapiGetAllDevices(@dataBuffer,sizeof(dataBuffer),neededsize,errmsg)<0 then
begin
    writeln(errmsg);
    halt;
end;

// gets device info from each handle
for i:=0 to neededsize div sizeof(integer)-1 do
begin
    if (apiGetDeviceInfo(dataBuffer[i], devinfos, errmsg)<0) then
    begin
        writeln(errmsg);
        halt;
    end;
    writeln(pansichar(@devinfos.serial)+' ('+pansichar(@devinfos.productname)+' ');
end;

end.

```

VB6 and yapi.dll

Each entry point from the yapi.dll is duplicated. You will find one regular C-decl version and one Visual Basic 6 compatible version, prefixed with `vb6_`.

20.4. 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.

21. High-level API Reference

This chapter summarizes the high-level API functions to drive your Yocto-CO2. 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.

21.1. General functions

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

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

js	<code><script type='text/javascript' src='yocto_api.js'></script></code>
cpp	<code>#include "yocto_api.h"</code>
m	<code>#import "yocto_api.h"</code>
pas	<code>uses yocto_api;</code>
vb	<code>yocto_api.vb</code>
cs	<code>yocto_api.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YModule;</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YModule;</code>
py	<code>from yocto_api import *</code>
php	<code>require_once('yocto_api.php');</code>
es	in HTML: <code><script src='../lib/yocto_api.js'></script></code> in node.js: <code>require('yoctolib-es2017/yocto_api.js');</code>

Global functions

yCheckLogicalName(name)

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

yClearHTTPCallbackCacheDir(bool_removeFiles)

Disables the HTTP callback cache.

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.

yGetCacheValidity()

Returns the validity period of the data loaded by the library.

yGetDeviceListValidity()

Returns the time between each forced enumeration of the YoctoHub used.

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 <code>RegisterHub()</code> .
<code>yRegisterDeviceArrivalCallback(arrivalCallback)</code>	Register a callback function, to be called each time a device is plugged.
<code>yRegisterDeviceRemovalCallback(removalCallback)</code>	Register a callback function, to be called each time a device is unplugged.
<code>yRegisterHub(url, errmsg)</code>	Setup the Yoctopuce library to use modules connected on a given machine.
<code>yRegisterHubDiscoveryCallback(hubDiscoveryCallback)</code>	Register a callback function, to be called each time an Network Hub send an SSDP message.
<code>yRegisterHubWebsocketCallback(ws, errmsg, authpwd)</code>	Variant to <code>RegisterHub()</code> used to initialize Yoctopuce API on an existing Websocket session, as happens for incoming websocket callbacks.
<code>yRegisterLogFunction(logfun)</code>	Registers a log callback function.
<code>ySelectArchitecture(arch)</code>	Select the architecture or the library to be loaded to access to USB.
<code>ySetCacheValidity(cacheValidityMs)</code>	Change the validity period of the data loaded by the library.
<code>ySetDelegate(object)</code>	(Objective-C only) Register an object that must follow the protocol <code>YDeviceHotPlug</code> .
<code>ySetDeviceListValidity(deviceListValidity)</code>	Change the time between each forced enumeration of the YoctoHub used.
<code>ySetHTTPCallbackCacheDir(str_directory)</code>	Enables the HTTP callback cache.
<code>ySetTimeout(callback, ms_timeout, args)</code>	Invoke the specified callback function after a given timeout.
<code>ySetUSBPacketAckMs(pktAckDelay)</code>	Enables the acknowledge of every USB packet received by the Yoctopuce library.
<code>ySleep(ms_duration, errmsg)</code>	Pauses the execution flow for a specified duration.
<code>yTestHub(url, mstimeout, errmsg)</code>	Test if the hub is reachable.
<code>yTriggerHubDiscovery(errmsg)</code>	Force a hub discovery, if a callback as been registered with <code>yRegisterHubDiscoveryCallback</code> it will be called for each net work hub that will respond to the discovery.
<code>yUnregisterHub(url)</code>	Setup the Yoctopuce library to no more use modules connected on a previously registered machine with <code>RegisterHub</code> .
<code>yUpdateDeviceList(errmsg)</code>	Triggers a (re)detection of connected Yoctopuce modules.
<code>yUpdateDeviceList_async(callback, context)</code>	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)
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)
uwp	bool CheckLogicalName (string name)
py	def CheckLogicalName (name)
php	function yCheckLogicalName (\$name)
es	function CheckLogicalName (name)

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

Parameters :

name a string containing the name to check.

Returns :

true if the name is valid, false otherwise.

YAPI.ClearHTTPCallbackCacheDir() yClearHTTPCallbackCacheDir()

YAPI

Disables the HTTP callback cache.

```
php function yClearHTTPCallbackCacheDir( $bool_removeFiles)
```

This method disables the HTTP callback cache, and can additionally cleanup the cache directory.

Parameters :

bool_removeFiles True to clear the content of the cache.

Returns :

nothing.

YAPI.DisableExceptions() yDisableExceptions()

YAPI

Disables the use of exceptions to report runtime errors.

js	function yDisableExceptions ()
cpp	void yDisableExceptions ()
m	+(void) DisableExceptions
pas	procedure yDisableExceptions ()
vb	procedure yDisableExceptions ()
cs	void DisableExceptions ()
uwp	void DisableExceptions ()
py	def DisableExceptions ()
php	function yDisableExceptions ()
es	function 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 ()
cpp	void yEnableExceptions ()
m	+(void) EnableExceptions
pas	procedure yEnableExceptions ()
vb	procedure yEnableExceptions ()
cs	void EnableExceptions ()
uwp	void EnableExceptions ()
py	def EnableExceptions ()
php	function yEnableExceptions ()
es	function EnableExceptions ()

Be aware that 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 ()
cpp	void yFreeAPI ()
m	+(void) FreeAPI
pas	procedure yFreeAPI ()
vb	procedure yFreeAPI ()
cs	void FreeAPI ()
java	void FreeAPI ()
uwp	void FreeAPI ()
py	def FreeAPI ()
php	function yFreeAPI ()
es	function 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 ()
cpp	string yGetAPIVersion ()
m	+(NSString*) GetAPIVersion
pas	function yGetAPIVersion (): string
vb	function yGetAPIVersion () As String
cs	String GetAPIVersion ()
java	String GetAPIVersion ()
uwp	string GetAPIVersion ()
py	def GetAPIVersion ()
php	function yGetAPIVersion ()
es	function GetAPIVersion ()

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

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

Returns :

a character string describing the library version.

YAPI.GetCacheValidity() yGetCacheValidity()

YAPI

Returns the validity period of the data loaded by the library.

cpp	static u64 yGetCacheValidity ()
m	+(u64) GetCacheValidity
pas	function yGetCacheValidity (): u64
vb	function yGetCacheValidity () As Long
cs	ulong GetCacheValidity ()
java	long GetCacheValidity ()
uwp	Task<ulong> GetCacheValidity ()
py	def GetCacheValidity ()
php	function yGetCacheValidity ()
es	function GetCacheValidity ()

This method returns the cache validity of all attributes module functions. Note: This function must be called after `yInitAPI` .

Returns :

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

YAPI.GetDeviceListValidity() yGetDeviceListValidity()

YAPI

Returns the time between each forced enumeration of the YoctoHub used.

cpp	static int yGetDeviceListValidity ()
m	+(int) GetDeviceListValidity
pas	function yGetDeviceListValidity (): LongInt
vb	function yGetDeviceListValidity () As Integer
cs	int GetDeviceListValidity ()
java	int GetDeviceListValidity ()
uwp	Task<int> GetDeviceListValidity ()
py	def GetDeviceListValidity ()
php	function yGetDeviceListValidity ()
es	function GetDeviceListValidity ()

Note: This function must be called after `yInitAPI`.

Returns :

the number of seconds between each enumeration.

YAPI.GetTickCount() yGetTickCount()

YAPI

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

js	function yGetTickCount ()
cpp	u64 yGetTickCount ()
m	+(u64) GetTickCount
pas	function yGetTickCount (): u64
vb	function yGetTickCount () As Long
cs	ulong GetTickCount ()
java	long GetTickCount ()
uwp	ulong GetTickCount ()
py	def GetTickCount ()
php	function yGetTickCount ()
es	function 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)
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 ()
uwp	Task<int> HandleEvents ()
py	def HandleEvents (errmsg =None)
php	function yHandleEvents (&\$ errmsg)
es	function HandleEvents (errmsg)

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

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

Parameters :

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

Returns :

YAPI_SUCCESS when the call succeeds.

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

YAPI.InitAPI() yInitAPI()

YAPI

Initializes the Yoctopuce programming library explicitly.

js	function yInitAPI (mode , errmsg)
cpp	YRETCODE yInitAPI (int mode , string& errmsg)
m	+(YRETCODE) InitAPI :(int) mode :(NSError**) errmsg
pas	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)
uwp	Task<int> InitAPI (int mode)
py	def InitAPI (mode , errmsg =None)
php	function yInitAPI (\$ mode , &\$ errmsg)
es	function InitAPI (mode , errmsg)

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

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

Parameters :

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

YAPI.PreregisterHub() yPreregisterHub()

Fault-tolerant alternative to RegisterHub().

js	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)
uwp	Task<int> PreregisterHub (string url)
py	def PreregisterHub (url , errmsg =None)
php	function yPreregisterHub (\$url, &\$errmsg)
es	function PreregisterHub (url , errmsg)

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)
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)
uwp	void RegisterDeviceArrivalCallback (DeviceUpdateHandler arrivalCallback)
py	def RegisterDeviceArrivalCallback (arrivalCallback)
php	function yRegisterDeviceArrivalCallback (\$arrivalCallback)
es	function 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()

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

js	function yRegisterDeviceRemovalCallback (removalCallback)
cpp	void yRegisterDeviceRemovalCallback (yDeviceUpdateCallback removalCallback)
m	+(void) RegisterDeviceRemovalCallback :(yDeviceUpdateCallback) removalCallback
pas	procedure yRegisterDeviceRemovalCallback (removalCallback : yDeviceUpdateFunc)
vb	procedure yRegisterDeviceRemovalCallback (ByVal removalCallback As yDeviceUpdateFunc)
cs	void RegisterDeviceRemovalCallback (yDeviceUpdateFunc removalCallback)
java	void RegisterDeviceRemovalCallback (DeviceRemovalCallback removalCallback)
uwp	void RegisterDeviceRemovalCallback (DeviceUpdateHandler removalCallback)
py	def RegisterDeviceRemovalCallback (removalCallback)
php	function yRegisterDeviceRemovalCallback (\$removalCallback)
es	function 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)
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)
uwp	Task<int> RegisterHub (string url)
py	def RegisterHub (url , errmsg =None)
php	function yRegisterHub (\$ url , &\$ errmsg)
es	function RegisterHub (url , errmsg)

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

usb: When the **usb** keyword is used, the API will work with devices connected directly to the USB bus. Some programming languages such 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)
uwp	Task RegisterHubDiscoveryCallback (HubDiscoveryHandler 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, the serial

YAPI.RegisterHubWebsocketCallback() yRegisterHubWebsocketCallback()

YAPI

Variant to RegisterHub() used to initialize Yoctopuce API on an existing Websocket session, as happens for incoming websocket callbacks.

Parameters :

- ws** node WebSocket object for the incoming websocket callback connection
- errmsg** a string passed by reference to receive any error message.
- authpwd** the optional authentication password, required only authentication is configured on the calling hub.

Returns :

YAPI_SUCCESS when the call succeeds.

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

YAPI.RegisterLogFunction() 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)
uwp	void RegisterLogFunction (LogHandler 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 not possible to reliably distinguish between a Hard Float (armhf) and a Soft Float (armel) install. For in this case, it is therefore recommended to manually select the proper architecture by calling `SelectArchitecture( )` before any other call to the library.

Parameters :

arch A string containing the architecture to use. Possibles value are: "armhf","armel", "i386","x86_64","32bit", "64bit"

Returns :

nothing.

On failure, throws an exception.

YAPI.SetCacheValidity() ySetCacheValidity()

YAPI

Change the validity period of the data loaded by the library.

cpp	static void ySetCacheValidity (u64 cacheValidityMs)
m	+(void) SetCacheValidity : (u64) cacheValidityMs
pas	procedure ySetCacheValidity (cacheValidityMs : u64)
vb	procedure ySetCacheValidity ()
cs	void SetCacheValidity (ulong cacheValidityMs)
java	void SetCacheValidity (long cacheValidityMs)
uwp	Task SetCacheValidity (ulong cacheValidityMs)
py	def SetCacheValidity (cacheValidityMs)
php	function ySetCacheValidity (\$cacheValidityMs)
es	function SetCacheValidity (cacheValidityMs)

By default, when accessing a module, all the attributes of the module functions are automatically kept in cache for the standard duration (5 ms). This method can be used to change this standard duration, for example in order to reduce network or USB traffic. This parameter does not affect value change callbacks Note: This function must be called after `yInitAPI`.

Parameters :

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

YAPI.SetDelegate() 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.SetDeviceListValidity() ySetDeviceListValidity()

YAPI

Change the time between each forced enumeration of the YoctoHub used.

cpp	static void ySetDeviceListValidity (int deviceListValidity)
m	+(void) SetDeviceListValidity : (int) deviceListValidity
pas	procedure ySetDeviceListValidity (deviceListValidity : LongInt)
vb	procedure ySetDeviceListValidity ()
cs	void SetDeviceListValidity (int deviceListValidity)
java	void SetDeviceListValidity (int deviceListValidity)
uwp	Task SetDeviceListValidity (int deviceListValidity)
py	def SetDeviceListValidity (deviceListValidity)
php	function ySetDeviceListValidity (\$ deviceListValidity)
es	function SetDeviceListValidity (deviceListValidity)

By default, the library performs a complete enumeration every 10 seconds. To reduce network traffic it is possible to increase this delay. This is particularly useful when a YoctoHub is connected to a GSM network where the traffic is charged. This setting does not affect modules connected by USB, nor the operation of arrival/removal callbacks. Note: This function must be called after `yInitAPI`.

Parameters :

deviceListValidity number of seconds between each enumeration.

YAPI.SetHTTPCallbackCacheDir() ySetHTTPCallbackCacheDir()

YAPI

Enables the HTTP callback cache.

```
php function ySetHTTPCallbackCacheDir( $str_directory)
```

When enabled, this cache reduces the quantity of data sent to the PHP script by 50% to 70%. To enable this cache, the method `ySetHTTPCallbackCacheDir()` must be called before any call to `yRegisterHub()`. This method takes in parameter the path of the directory used for saving data between each callback. This folder must exist and the PHP script needs to have write access to it. It is recommended to use a folder that is not published on the Web server since the library will save some data of Yoctopuce devices into this folder.

Note: This feature is supported by YoctoHub and VirtualHub since version 27750.

Parameters :

str_directory the path of the folder that will be used as cache.

Returns :

nothing.

On failure, throws an exception.

YAPI.SetTimeout() ySetTimeout()

YAPI

Invoke the specified callback function after a given timeout.

```
js function ySetTimeout( callback, ms_timeout, args)
```

```
es function SetTimeout( callback, ms_timeout, args)
```

This function behaves more or less like Javascript `setTimeout`, but during the waiting time, it will call `yHandleEvents` and `yUpdateDeviceList` periodically, in order to keep the API up-to-date with current devices.

Parameters :

- callback** the function to call after the timeout occurs. On Microsoft Internet Explorer, the callback must be provided as a string to be evaluated.
- ms_timeout** an integer corresponding to the duration of the timeout, in milliseconds.
- args** additional arguments to be passed to the callback function can be provided, if needed (not supported on Microsoft Internet Explorer).

Returns :

YAPI_SUCCESS when the call succeeds.

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

YAPI.SetUSBPacketAckMs() ySetUSBPacketAckMs()

YAPI

Enables the acknowledge of every USB packet received by the Yoctopuce library.

```
java void SetUSBPacketAckMs( int pktAckDelay)
```

This function allows the library to run on Android phones that tend to loose USB packets. By default, this feature is disabled because it doubles the number of packets sent and slows down the API considerably. Therefore, the acknowledge of incoming USB packets should only be enabled on phones or tablets that loose USB packets. A delay of 50 milliseconds is generally enough. In case of doubt, contact Yoctopuce support. To disable USB packets acknowledge, call this function with the value 0. Note: this feature is only available on Android.

Parameters :

pktAckDelay then number of milliseconds before the module

YAPI.Sleep() ySleep()

YAPI

Pauses the execution flow for a specified duration.

js	function ySleep (ms_duration , errmsg)
cpp	YRETCODE ySleep (unsigned ms_duration , string& errmsg)
m	+(YRETCODE) Sleep :(unsigned) ms_duration :(NSError **) errmsg
pas	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)
uwp	Task<int> Sleep (ulong ms_duration)
py	def Sleep (ms_duration , errmsg =None)
php	function ySleep (\$ms_duration , & \$errmsg)
es	function Sleep (ms_duration , errmsg)

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

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

Parameters :

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

Returns :

YAPI_SUCCESS when the call succeeds.

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

YAPI.TestHub() yTestHub()

Test if the hub is reachable.

```

cpp YRETCODE yTestHub( const string& url, int mstimeout, string& errmsg)
m   +(YRETCODE) TestHub : (NSString*) url
                                : (int) mstimeout
                                : (NSError**) errmsg
pas function yTestHub( url: string,
                        mstimeout: integer,
                        var errmsg: string): integer
vb   function yTestHub( ByVal url As String,
                        ByVal mstimeout As Integer,
                        ByRef errmsg As String) As Integer
cs   int TestHub( string url, int mstimeout, ref string errmsg)
java int TestHub( String url, int mstimeout)
uwp  Task<int> TestHub( string url, uint mstimeout)
py   def TestHub( url, mstimeout, errmsg=None)
php  function yTestHub( $url, $mstimeout, &$errmsg)
es   function TestHub( url, mstimeout, errmsg)

```

This method do not register the hub, it only test if the hub is usable. The url parameter follow the same convention as the RegisterHub method. This method is useful to verify the authentication parameters for a hub. It is possible to force this method to return after mstimeout milliseconds.

Parameters :

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

Returns :

YAPI_SUCCESS when the call succeeds.

On failure returns a negative error code.

YAPI.TriggerHubDiscovery() yTriggerHubDiscovery()

YAPI

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

cpp	YRETCODE yTriggerHubDiscovery(string& errmsg)
m	+(YRETCODE) TriggerHubDiscovery : (NSError**) errmsg
pas	function yTriggerHubDiscovery(var errmsg: string): integer
vb	function yTriggerHubDiscovery(ByRef errmsg As String) As Integer
cs	int TriggerHubDiscovery(ref string errmsg)
java	int TriggerHubDiscovery()
uwp	Task<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)
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)
uwp	Task UnregisterHub (string url)
py	def UnregisterHub (url)
php	function yUnregisterHub (\$url)
es	function UnregisterHub (url)

Parameters :

url a string containing either "usb" or the

YAPI.UpdateDeviceList() yUpdateDeviceList()

YAPI

Triggers a (re)detection of connected Yoctopuce modules.

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

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

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

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)
```

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.

21.2. 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	<code><script type='text/javascript' src='yocto_api.js'></script></code>
cpp	<code>#include "yocto_api.h"</code>
m	<code>#import "yocto_api.h"</code>
pas	<code>uses yocto_api;</code>
vb	<code>yocto_api.vb</code>
cs	<code>yocto_api.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YModule;</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YModule;</code>
py	<code>from yocto_api import *</code>
php	<code>require_once('yocto_api.php');</code>
es	in HTML: <code><script src=" ../lib/yocto_api.js"></script></code> in node.js: <code>require('yoctolib-es2017/yocto_api.js');</code>

Global functions

yFindModule(func)

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

yFindModuleInContext(yctx, func)

Retrieves a module for a given identifier in a YAPI context.

yFirstModule()

Starts the enumeration of modules currently accessible.

YModule methods

module→checkFirmware(path, onlynew)

Tests whether the byn file is valid for this module.

module→clearCache()

Invalidates the cache.

module→describe()

Returns a descriptive text that identifies the module.

module→download(pathname)

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

module→functionBaseType(functionIndex)

Retrieves the base type of the *n*th function on the module.

module→functionCount()

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

module→functionId(functionIndex)

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

module→functionName(functionIndex)

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

module→functionType(functionIndex)

Retrieves the type of the *n*th function on the module.

module→functionValue(functionIndex)

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

module→get_allSettings()

Returns all the settings and uploaded files of the module.

module→get_beacon()

Returns the state of the localization beacon.

module→get_errorMessage()

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

module→get_errorType()

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

module→get_firmwareRelease()

Returns the version of the firmware embedded in the module.

module→get_functionIds(funType)

Retrieve all hardware identifier that match the type passed in argument.

module→get_hardwareId()

Returns the unique hardware identifier of the module.

module→get_icon2d()

Returns the icon of the module.

module→get_lastLogs()

Returns a string with last logs of the module.

module→get_logicalName()

Returns the logical name of the module.

module→get_luminosity()

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

module→get_parentHub()

Returns the serial number of the YoctoHub on which this module is connected.

module→get_persistentSettings()

Returns the current state of persistent module settings.

module→get_productId()

Returns the USB device identifier of the module.

module→get_productName()

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

module→get_productRelease()

Returns the 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_subDevices()

Returns a list of all the modules that are plugged into the current module.

module→get_upTime()

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

module→get_url()

Returns the URL used to access the module.

module→get_usbCurrent()

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

module→get_userData()

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

module→get_userVar()

Returns the value previously stored in this attribute.

module→hasFunction(funcId)

Tests if the device includes a specific function.

module→isOnline()

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

module→isOnline_async(callback, context)

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

module→load(msValidity)

Preloads the module cache with a specified validity duration.

module→load_async(msValidity, callback, context)

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

module→log(text)

Adds a text message to the device logs.

module→nextModule()

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

module→reboot(secBeforeReboot)

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

module→registerBeaconCallback(callback)

Register a callback function, to be called when the localization beacon of the module has been changed.

module→registerConfigChangeCallback(callback)

Register a callback function, to be called when a persistent settings in a device configuration has been changed (e.g.

module→registerLogCallback(callback)

Registers a device log callback function.

module→revertFromFlash()

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

module→saveToFlash()

Saves current settings in the nonvolatile memory of the module.

module→set_allSettings(settings)

Restores all the settings of the device.

module→set_allSettingsAndFiles(settings)

Restores all the settings and uploaded files to the module.

module→set_beacon(newval)

Turns on or off the module localization beacon.

module→set_logicalName(newval)

Changes the logical name of the module.

module→set_luminosity(newval)

Changes the luminosity of the module informative leds.

module→set_userData(data)

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

module→set_userVar(newval)

Stores a 32 bit value in the device RAM.

module→triggerConfigChangeCallback()

Triggers a configuration change callback, to check if they are supported or not.

module→**triggerFirmwareUpdate(secBeforeReboot)**

Schedules a module reboot into special firmware update mode.

module→**updateFirmware(path)**

Prepares a firmware update of the module.

module→**updateFirmwareEx(path, force)**

Prepares a firmware update of the module.

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

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

YModule.FindModule() yFindModule()

YModule

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

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

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

If a call to this object's `is_online()` method returns FALSE although you are certain that the device is plugged, make sure that you did call `registerHub()` at application initialization time.

Parameters :

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

Returns :

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

YModule.FindModuleInContext() yFindModuleInContext()

YModule

Retrieves a module for a given identifier in a YAPI context.

java	YModule FindModuleInContext (YAPIContext yctx , String func)
uwp	YModule FindModuleInContext (YAPIContext yctx , string func)
es	function FindModuleInContext (yctx , func)

The identifier can be specified using several formats:

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

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

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the module

Returns :

a `YModule` object allowing you to drive the module.

YModule.FirstModule() yFirstModule()

YModule

Starts the enumeration of modules currently accessible.

js	function yFirstModule ()
cpp	YModule* yFirstModule ()
m	+(YModule*) FirstModule
pas	function yFirstModule (): TModule
vb	function yFirstModule () As YModule
cs	YModule FirstModule ()
java	YModule FirstModule ()
uwp	YModule FirstModule ()
py	def FirstModule ()
php	function yFirstModule ()
es	function 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**

Tests whether the byn file is valid for this module.

js	function checkFirmware (path , onlynew)
cpp	string checkFirmware (string path , bool onlynew)
m	-(NSString*) checkFirmware : (NSString*) path : (bool) onlynew
pas	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)
uwp	Task<string> checkFirmware (string path , bool onlynew)
py	def checkFirmware (path , onlynew)
php	function checkFirmware (\$ path , \$ onlynew)
es	function checkFirmware (path , onlynew)
cmd	YModule target checkFirmware path onlynew

This method is useful to test if the module needs to be updated. It is possible to pass a directory as argument instead of a file. In this case, this method returns the path of the most recent appropriate .byn file. If the parameter `onlynew` is true, the function discards firmwares that are older or equal to the installed firmware.

Parameters :

path the path of a byn file or a directory that contains byn files
onlynew returns only files that are strictly newer

Returns :

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

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

module→clearCache()**YModule**

Invalidates the cache.

js	function clearCache ()
cpp	void clearCache ()
m	-(void) clearCache
pas	procedure clearCache ()
vb	procedure clearCache ()
cs	void clearCache ()
java	void clearCache ()
py	def clearCache ()
php	function clearCache ()
es	function clearCache ()

Invalidates the cache of the module attributes. Forces the next call to `get_xxx()` or `loadxxx()` to use values that come from the device.

module→**describe()****YModule**

Returns a descriptive text that identifies the module.

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

The text may include either the logical name or the serial number of the module.

Returns :

a string that describes the module

module→download()**YModule**

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

js	function download (pathname)
cpp	string download (string pathname)
m	-(NSMutableData*) download : (NSString*) pathname
pas	function download (pathname : string): TByteArray
vb	function download () As Byte
cs	byte[] download (string pathname)
java	byte[] download (String pathname)
uwp	Task<byte[]> download (string pathname)
py	def download (pathname)
php	function download (\$ pathname)
es	function download (pathname)
cmd	YModule target download pathname

Parameters :

pathname name of the new file to load

Returns :

a binary buffer with the file content

On failure, throws an exception or returns YAPI_INVALID_STRING.

module→functionBaseType()**YModule**

Retrieves the base type of the *n*th function on the module.

js	function functionBaseType (functionIndex)
cpp	string functionBaseType (int functionIndex)
pas	function functionBaseType (functionIndex : integer): string
vb	function functionBaseType (ByVal functionIndex As Integer) As String
cs	string functionBaseType (int functionIndex)
java	String functionBaseType (int functionIndex)
py	def functionBaseType (functionIndex)
php	function functionBaseType (\$functionIndex)
es	function functionBaseType (functionIndex)

For instance, the base type of all measuring functions is "Sensor".

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a string corresponding to the base type of the function

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

module→functionCount()**YModule**

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

js	function functionCount ()
cpp	int functionCount ()
m	-(int) functionCount
pas	function functionCount (): integer
vb	function functionCount () As Integer
cs	int functionCount ()
java	int functionCount ()
py	def functionCount ()
php	function functionCount ()
es	function functionCount ()

Returns :

the number of functions on the module

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

module→functionId()**YModule**

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

js	function functionId (functionIndex)
cpp	string functionId (int functionIndex)
m	-(NSString*) functionId : (int) functionIndex
pas	function functionId (functionIndex : integer): string
vb	function functionId (ByVal functionIndex As Integer) As String
cs	string functionId (int functionIndex)
java	String functionId (int functionIndex)
py	def functionId (functionIndex)
php	function functionId (\$functionIndex)
es	function functionId (functionIndex)

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a string corresponding to the unambiguous hardware identifier of the requested module function

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

module→functionName()**YModule**

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

js	function functionName (functionIndex)
cpp	string functionName (int functionIndex)
m	-(NSString*) functionName : (int) functionIndex
pas	function functionName (functionIndex : integer): string
vb	function functionName (ByVal functionIndex As Integer) As String
cs	string functionName (int functionIndex)
java	String functionName (int functionIndex)
py	def functionName (functionIndex)
php	function functionName (\$functionIndex)
es	function functionName (functionIndex)

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a string corresponding to the logical name of the requested module function

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

module→functionType()**YModule**

Retrieves the type of the *n*th function on the module.

js	function functionType (functionIndex)
cpp	string functionType (int functionIndex)
pas	function functionType (functionIndex : integer): string
vb	function functionType (ByVal functionIndex As Integer) As String
cs	string functionType (int functionIndex)
java	String functionType (int functionIndex)
py	def functionType (functionIndex)
php	function functionType (\$functionIndex)
es	function functionType (functionIndex)

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a string corresponding to the type of the function

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

module→functionValue()**YModule**

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

js	function functionValue (functionIndex)
cpp	string functionValue (int functionIndex)
m	-(NSString*) functionValue : (int) functionIndex
pas	function functionValue (functionIndex : integer): string
vb	function functionValue (ByVal functionIndex As Integer) As String
cs	string functionValue (int functionIndex)
java	String functionValue (int functionIndex)
py	def functionValue (functionIndex)
php	function functionValue (\$functionIndex)
es	function functionValue (functionIndex)

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a short string (up to 6 characters) corresponding to the advertised value of the requested module function

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

module→get_allSettings()**YModule****module→allSettings()**

Returns all the settings and uploaded files of the module.

js	function get_allSettings ()
cpp	string get_allSettings ()
m	-(NSMutableData*) allSettings
pas	function get_allSettings (): TByteArray
vb	function get_allSettings () As Byte
cs	byte[] get_allSettings ()
java	byte[] get_allSettings ()
uwp	Task<byte[]> get_allSettings ()
py	def get_allSettings ()
php	function get_allSettings ()
es	function get_allSettings ()
cmd	YModule target get_allSettings

Useful to backup all the logical names, calibrations parameters, and uploaded files of a device.

Returns :

a binary buffer with all the settings.

On failure, throws an exception or returns an binary object of size 0.

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

Returns the state of the localization beacon.

js	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 ()
uwp	Task<int> get_beacon ()
py	def get_beacon ()
php	function get_beacon ()
es	function get_beacon ()
cmd	YModule target get_beacon

Returns :

either Y_BEACON_OFF or Y_BEACON_ON, according to the state of the localization beacon

On failure, throws an exception or returns Y_BEACON_INVALID.

module→**get_errorMessage()****YModule****module**→**errorMessage()**

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

js	function get_errorMessage ()
cpp	string get_errorMessage ()
m	-(NSString*) errorMessage
pas	function get_errorMessage (): string
vb	function get_errorMessage () As String
cs	string get_errorMessage ()
java	String get_errorMessage ()
py	def get_errorMessage ()
php	function get_errorMessage ()
es	function 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()****YModule****module**→**errorType()**

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

js	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 ()
php	function get_errorType ()
es	function get_errorType ()

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

Returns :

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

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

Returns the version of the firmware embedded in the module.

js	function get_firmwareRelease ()
cpp	string get_firmwareRelease ()
m	-(NSString*) firmwareRelease
pas	function get_firmwareRelease (): string
vb	function get_firmwareRelease () As String
cs	string get_firmwareRelease ()
java	String get_firmwareRelease ()
uwp	Task<string> get_firmwareRelease ()
py	def get_firmwareRelease ()
php	function get_firmwareRelease ()
es	function get_firmwareRelease ()
cmd	YModule target get_firmwareRelease

Returns :

a string corresponding to the version of the firmware embedded in the module

On failure, throws an exception or returns Y_FIRMWARERELEASE_INVALID.

module→**get_functionIds()****YModule****module**→**functionIds()**

Retrieve all hardware identifier that match the type passed in argument.

js	function get_functionIds (funType)
cpp	vector<string> get_functionIds (string funType)
m	-(NSMutableArray*) functionIds : (NSString*) funType
pas	function get_functionIds (funType : string): TStringArray
vb	function get_functionIds () As List
cs	List<string> get_functionIds (string funType)
java	ArrayList<String> get_functionIds (String funType)
uwp	Task<List<string>> get_functionIds (string funType)
py	def get_functionIds (funType)
php	function get_functionIds (\$funType)
es	function get_functionIds (funType)
cmd	YModule target get_functionIds funType

Parameters :

funType The type of function (Relay, LightSensor, Voltage,...)

Returns :

an array of strings.

module→**get_hardwareId()****YModule****module**→**hardwareId()**

Returns the unique hardware identifier of the module.

js	function get_hardwareId ()
cpp	string get_hardwareId ()
m	-(NSString*) hardwareId
vb	function get_hardwareId () As String
cs	string get_hardwareId ()
java	String get_hardwareId ()
py	def get_hardwareId ()
php	function get_hardwareId ()
es	function 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 ()
cpp	string get_icon2d ()
m	-(NSMutableData*) icon2d
pas	function get_icon2d (): TByteArray
vb	function get_icon2d () As Byte
cs	byte[] get_icon2d ()
java	byte[] get_icon2d ()
uwp	Task<byte[]> get_icon2d ()
py	def get_icon2d ()
php	function get_icon2d ()
es	function get_icon2d ()
cmd	YModule target get_icon2d

The icon is a PNG image and does not exceeds 1536 bytes.

Returns :

a binary buffer with module icon, in png format. On failure, throws an exception or returns YAPI_INVALID_STRING.

module→get_lastLogs()**YModule****module→lastLogs()**

Returns a string with last logs of the module.

js	function get_lastLogs ()
cpp	string get_lastLogs ()
m	-(NSString*) lastLogs
pas	function get_lastLogs (): string
vb	function get_lastLogs () As String
cs	string get_lastLogs ()
java	String get_lastLogs ()
uwp	Task<string> get_lastLogs ()
py	def get_lastLogs ()
php	function get_lastLogs ()
es	function 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 ()
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 ()
uwp	Task<string> get_logicalName ()
py	def get_logicalName ()
php	function get_logicalName ()
es	function 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()**YModule****module→luminosity()**

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

js	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 ()
uwp	Task<int> get_luminosity ()
py	def get_luminosity ()
php	function get_luminosity ()
es	function get_luminosity ()
cmd	YModule target get_luminosity

Returns :

an integer corresponding to the luminosity of the module informative leds (from 0 to 100)

On failure, throws an exception or returns Y_LUMINOSITY_INVALID.

module→**get_parentHub()****YModule****module**→**parentHub()**

Returns the serial number of the YoctoHub on which this module is connected.

js	function get_parentHub ()
cpp	string get_parentHub ()
m	-(NSString*) parentHub
pas	function get_parentHub (): string
vb	function get_parentHub () As String
cs	string get_parentHub ()
java	String get_parentHub ()
uwp	Task<string> get_parentHub ()
py	def get_parentHub ()
php	function get_parentHub ()
es	function get_parentHub ()
cmd	YModule target get_parentHub

If the module is connected by USB, or if the module is the root YoctoHub, an empty string is returned.

Returns :

a string with the serial number of the YoctoHub or an empty string

module→get_persistentSettings()**YModule****module→persistentSettings()**

Returns the current state of persistent module settings.

js	function get_persistentSettings ()
cpp	Y_PERSISTENTSETTINGS_enum get_persistentSettings ()
m	-(Y_PERSISTENTSETTINGS_enum) persistentSettings
pas	function get_persistentSettings (): Integer
vb	function get_persistentSettings () As Integer
cs	int get_persistentSettings ()
java	int get_persistentSettings ()
uwp	Task<int> get_persistentSettings ()
py	def get_persistentSettings ()
php	function get_persistentSettings ()
es	function 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()**YModule****module→productId()**

Returns the USB device identifier of the module.

js	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 ()
uwp	Task<int> get_productId ()
py	def get_productId ()
php	function get_productId ()
es	function get_productId ()
cmd	YModule target get_productId

Returns :

an integer corresponding to the USB device identifier of the module

On failure, throws an exception or returns Y_PRODUCTID_INVALID.

module→get_productName()**YModule****module→productName()**

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

js	function get_productName ()
cpp	string get_productName ()
m	-(NSString*) productName
pas	function get_productName (): string
vb	function get_productName () As String
cs	string get_productName ()
java	String get_productName ()
uwp	Task<string> get_productName ()
py	def get_productName ()
php	function get_productName ()
es	function get_productName ()
cmd	YModule target get_productName

Returns :

a string corresponding to the commercial name of the module, as set by the factory

On failure, throws an exception or returns Y_PRODUCTNAME_INVALID.

module→**get_productRelease()****YModule****module**→**productRelease()**

Returns the hardware release version of the module.

js	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 ()
uwp	Task<int> get_productRelease ()
py	def get_productRelease ()
php	function get_productRelease ()
es	function 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 ()
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 ()
uwp	Task<int> get_rebootCountdown ()
py	def get_rebootCountdown ()
php	function get_rebootCountdown ()
es	function get_rebootCountdown ()
cmd	YModule target get_rebootCountdown

Returns :

an integer corresponding to the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled

On failure, throws an exception or returns Y_REBOOTCOUNTDOWN_INVALID.

module→**get_serialNumber()****YModule****module**→**serialNumber()**

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

js	function get_serialNumber ()
cpp	string get_serialNumber ()
m	-(NSString*) serialNumber
pas	function get_serialNumber (): string
vb	function get_serialNumber () As String
cs	string get_serialNumber ()
java	String get_serialNumber ()
uwp	Task<string> get_serialNumber ()
py	def get_serialNumber ()
php	function get_serialNumber ()
es	function get_serialNumber ()
cmd	YModule target get_serialNumber

Returns :

a string corresponding to the serial number of the module, as set by the factory

On failure, throws an exception or returns Y_SERIALNUMBER_INVALID.

module→**get_subDevices()****YModule****module**→**subDevices()**

Returns a list of all the modules that are plugged into the current module.

js	function get_subDevices ()
cpp	vector<string> get_subDevices ()
m	-(NSMutableArray*) subDevices
pas	function get_subDevices (): TStringArray
vb	function get_subDevices () As List
cs	List<string> get_subDevices ()
java	ArrayList<String> get_subDevices ()
uwp	Task<List<string>> get_subDevices ()
py	def get_subDevices ()
php	function get_subDevices ()
es	function get_subDevices ()
cmd	YModule target get_subDevices

This method only makes sense when called for a YoctoHub/VirtualHub. Otherwise, an empty array will be returned.

Returns :

an array of strings containing the sub modules.

module→**get_upTime()****YModule****module**→**upTime()**

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

js	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 ()
uwp	Task<long> get_upTime ()
py	def get_upTime ()
php	function get_upTime ()
es	function get_upTime ()
cmd	YModule target get_upTime

Returns :

an integer corresponding to the number of milliseconds spent since the module was powered on

On failure, throws an exception or returns Y_UPTIME_INVALID.

module→**get_url()****YModule****module**→**url()**

Returns the URL used to access the module.

js	function get_url ()
cpp	string get_url ()
m	-(NSString*) url
pas	function get_url (): string
vb	function get_url () As String
cs	string get_url ()
java	String get_url ()
uwp	Task<string> get_url ()
py	def get_url ()
php	function get_url ()
es	function get_url ()
cmd	YModule target get_url

If the module is connected by USB, the string 'usb' is returned.

Returns :

a string with the URL of the module.

module→get_usbCurrent()**YModule****module→usbCurrent()**

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

js	function get_usbCurrent ()
cpp	int get_usbCurrent ()
m	-(int) usbCurrent
pas	function get_usbCurrent (): LongInt
vb	function get_usbCurrent () As Integer
cs	int get_usbCurrent ()
java	int get_usbCurrent ()
uwp	Task<int> get_usbCurrent ()
py	def get_usbCurrent ()
php	function get_usbCurrent ()
es	function 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()**YModule****module→userData()**

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

js	function get_userData ()
cpp	void * get_userData ()
m	-(id) userData
pas	function get_userData (): Tobject
vb	function get_userData () As Object
cs	object get_userData ()
java	Object get_userData ()
py	def get_userData ()
php	function get_userData ()
es	function get_userData ()

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

Returns :

the object stored previously by the caller.

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

Returns the value previously stored in this attribute.

js	function get_userVar ()
cpp	int get_userVar ()
m	-(int) userVar
pas	function get_userVar (): LongInt
vb	function get_userVar () As Integer
cs	int get_userVar ()
java	int get_userVar ()
uwp	Task<int> get_userVar ()
py	def get_userVar ()
php	function get_userVar ()
es	function get_userVar ()
cmd	YModule target get_userVar

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

Returns :

an integer corresponding to the value previously stored in this attribute

On failure, throws an exception or returns Y_USERVAR_INVALID.

module→hasFunction()**YModule**

Tests if the device includes a specific function.

js	function hasFunction (funcId)
cpp	bool hasFunction (string funcId)
m	-(bool) hasFunction : (NSString*) funcId
pas	function hasFunction (funcId : string): boolean
vb	function hasFunction () As Boolean
cs	bool hasFunction (string funcId)
java	boolean hasFunction (String funcId)
uwp	Task<bool> hasFunction (string funcId)
py	def hasFunction (funcId)
php	function hasFunction (\$funcId)
es	function hasFunction (funcId)
cmd	YModule target hasFunction funcId

This method takes a function identifier and returns a boolean.

Parameters :

funcId the requested function identifier

Returns :

true if the device has the function identifier

module→isOnline()**YModule**

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

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

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

Returns :

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

module→isOnline_async()**YModule**

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

```
js function isOnline_async( callback, context)
```

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox Javascript VM that does not implement context switching during blocking I/O calls.

Parameters :

- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving module object and the boolean result
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

module→load()**YModule**

Preloads the module cache with a specified validity duration.

js	function load (msValidity)
cpp	YRETCODE load (int msValidity)
m	-(YRETCODE) load : (u64) msValidity
pas	function load (msValidity : u64): YRETCODE
vb	function load (ByVal msValidity As Long) As YRETCODE
cs	YRETCODE load (ulong msValidity)
java	int load (long msValidity)
py	def load (msValidity)
php	function load (\$msValidity)
es	function load (msValidity)

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

Parameters :

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

Returns :

YAPI_SUCCESS when the call succeeds.

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

module→load_async()**YModule**

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

```
js function load_async( msValidity, callback, context)
```

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

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

- msValidity** an integer corresponding to the validity of the loaded module parameters, in milliseconds
- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving module object and the error code (or YAPI_SUCCESS)
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

module→log()**YModule**

Adds a text message to the device logs.

js	function log (text)
c++	int log (string text)
m	-(int) log : (NSString*) text
pas	function log (text : string): LongInt
vb	function log () As Integer
cs	int log (string text)
java	int log (String text)
uwp	Task<int> log (string text)
py	def log (text)
php	function log (\$ text)
es	function log (text)
cmd	YModule target log text

This function is useful in particular to trace the execution of HTTP callbacks. If a newline is desired after the message, it must be included in the string.

Parameters :

text the string to append to the logs.

Returns :

YAPI_SUCCESS if the call succeeds.

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

module→nextModule()**YModule**

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

js	function nextModule ()
cpp	YModule * nextModule ()
m	-(YModule*) nextModule
pas	function nextModule (): TYModule
vb	function nextModule () As YModule
cs	YModule nextModule ()
java	YModule nextModule ()
uwp	YModule nextModule ()
py	def nextModule ()
php	function nextModule ()
es	function 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)
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)
uwp	Task<int> reboot (int secBeforeReboot)
py	def reboot (secBeforeReboot)
php	function reboot (\$secBeforeReboot)
es	function reboot (secBeforeReboot)
cmd	YModule target reboot secBeforeReboot

Parameters :

secBeforeReboot number of seconds before rebooting

Returns :

YAPI_SUCCESS when the call succeeds.

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

module→registerBeaconCallback()**YModule**

Register a callback function, to be called when the localization beacon of the module has been changed.

js	function registerBeaconCallback (callback)
cpp	int registerBeaconCallback (YModuleBeaconCallback callback)
m	-(int) registerBeaconCallback : (YModuleBeaconCallback) callback
pas	function registerBeaconCallback (callback : TYModuleBeaconCallback): LongInt
vb	function registerBeaconCallback () As Integer
cs	int registerBeaconCallback (BeaconCallback callback)
java	int registerBeaconCallback (BeaconCallback callback)
uwp	Task<int> registerBeaconCallback (BeaconCallback callback)
py	def registerBeaconCallback (callback)
php	function registerBeaconCallback (\$callback)
es	function registerBeaconCallback (callback)

The callback function should take two arguments: the YModule object of which the beacon has changed, and an integer describing the new beacon state.

Parameters :

callback The callback function to call, or `null` to unregister a

module→registerConfigChangeCallback()**YModule**

Register a callback function, to be called when a persistent settings in a device configuration has been changed (e.g.

js	function registerConfigChangeCallback (callback)
cpp	int registerConfigChangeCallback (YModuleConfigChangeCallback callback)
m	-(int) registerConfigChangeCallback : (YModuleConfigChangeCallback) callback
pas	function registerConfigChangeCallback (callback : TYModuleConfigChangeCallback): LongInt
vb	function registerConfigChangeCallback () As Integer
cs	int registerConfigChangeCallback (ConfigChangeCallback callback)
java	int registerConfigChangeCallback (ConfigChangeCallback callback)
uwp	Task<int> registerConfigChangeCallback (ConfigChangeCallback callback)
py	def registerConfigChangeCallback (callback)
php	function registerConfigChangeCallback (\$callback)
es	function registerConfigChangeCallback (callback)

change of unit, etc).

Parameters :

callback a procedure taking a YModule parameter, or null

module→registerLogCallback()**YModule**

Registers a device log callback function.

cpp	<code>void registerLogCallback(YModuleLogCallback callback)</code>
m	<code>-(void) registerLogCallback : (YModuleLogCallback) callback</code>
vb	<code>function registerLogCallback(ByVal callback As YModuleLogCallback) As Integer</code>
cs	<code>int registerLogCallback(LogCallback callback)</code>
java	<code>void registerLogCallback(LogCallback callback)</code>
py	<code>def registerLogCallback(callback)</code>

This callback will be called each time that a module sends a new log message. Mostly useful to debug a Yoctopuce module.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the module object that emitted the log message, and the character string containing the log.

module→revertFromFlash()**YModule**

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

js	function revertFromFlash ()
cpp	int revertFromFlash ()
m	-(int) revertFromFlash
pas	function revertFromFlash (): LongInt
vb	function revertFromFlash () As Integer
cs	int revertFromFlash ()
java	int revertFromFlash ()
uwp	Task<int> revertFromFlash ()
py	def revertFromFlash ()
php	function revertFromFlash ()
es	function revertFromFlash ()
cmd	YModule target revertFromFlash

Returns :

YAPI_SUCCESS when the call succeeds.

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

module→saveToFlash()**YModule**

Saves current settings in the nonvolatile memory of the module.

js	function saveToFlash ()
cpp	int saveToFlash ()
m	-(int) saveToFlash
pas	function saveToFlash (): LongInt
vb	function saveToFlash () As Integer
cs	int saveToFlash ()
java	int saveToFlash ()
uwp	Task<int> saveToFlash ()
py	def saveToFlash ()
php	function saveToFlash ()
es	function saveToFlash ()
cmd	YModule target saveToFlash

Warning: the number of allowed save operations during a module life is limited (about 100000 cycles). Do not call this function within a loop.

Returns :

YAPI_SUCCESS when the call succeeds.

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

module→**set_allSettings()****YModule****module**→**setAllSettings()**

Restores all the settings of the device.

js	function set_allSettings (settings)
cpp	int set_allSettings (string settings)
m	-(int) setAllSettings : (NSData*) settings
pas	function set_allSettings (settings : TByteArray): LongInt
vb	procedure set_allSettings ()
cs	int set_allSettings ()
java	int set_allSettings (byte[] settings)
uwp	Task<int> set_allSettings ()
py	def set_allSettings (settings)
php	function set_allSettings (\$ settings)
es	function set_allSettings (settings)
cmd	YModule target set_allSettings settings

Useful to restore all the logical names and calibrations parameters of a module from a backup. Remember to call the `saveToFlash()` method of the module if the modifications must be kept.

Parameters :

settings a binary buffer with all the settings.

Returns :

YAPI_SUCCESS when the call succeeds.

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

module→set_allSettingsAndFiles()**YModule****module→setAllSettingsAndFiles()**

Restores all the settings and uploaded files to the module.

js	function set_allSettingsAndFiles (settings)
cpp	int set_allSettingsAndFiles (string settings)
m	-(int) setAllSettingsAndFiles : (NSData*) settings
pas	function set_allSettingsAndFiles (settings : TByteArray): LongInt
vb	procedure set_allSettingsAndFiles ()
cs	int set_allSettingsAndFiles ()
java	int set_allSettingsAndFiles (byte[] settings)
uwp	Task<int> set_allSettingsAndFiles ()
py	def set_allSettingsAndFiles (settings)
php	function set_allSettingsAndFiles (\$settings)
es	function set_allSettingsAndFiles (settings)
cmd	YModule target set_allSettingsAndFiles settings

This method is useful to restore all the logical names and calibrations parameters, uploaded files etc. of a device from a backup. Remember to call the `saveToFlash()` method of the module if the modifications must be kept.

Parameters :

settings a binary buffer with all the settings.

Returns :

YAPI_SUCCESS when the call succeeds.

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

module→**set_beacon()****YModule****module**→**setBeacon()**

Turns on or off the module localization beacon.

js	function set_beacon (newval)
cpp	int set_beacon (Y_BEACON_enum newval)
m	-(int) setBeacon : (Y_BEACON_enum) newval
pas	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)
uwp	Task<int> set_beacon (int newval)
py	def set_beacon (newval)
php	function set_beacon (\$ newval)
es	function set_beacon (newval)
cmd	YModule target set_beacon newval

Parameters :

newval either Y_BEACON_OFF or Y_BEACON_ON

Returns :

YAPI_SUCCESS if the call succeeds.

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

module→**set_logicalName()****YModule****module**→**setLogicalName()**

Changes the logical name of the module.

js	function set_logicalName (newval)
cpp	int set_logicalName (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)
uwp	Task<int> set_logicalName (string newval)
py	def set_logicalName (newval)
php	function set_logicalName (\$ newval)
es	function 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()****YModule****module**→**setLuminosity()**

Changes the luminosity of the module informative leds.

js	function set_luminosity (newval)
cpp	int set_luminosity (int newval)
m	-(int) setLuminosity : (int) newval
pas	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)
uwp	Task<int> set_luminosity (int newval)
py	def set_luminosity (newval)
php	function set_luminosity (\$newval)
es	function set_luminosity (newval)
cmd	YModule target set_luminosity newval

The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer corresponding to the luminosity of the module informative leds

Returns :

YAPI_SUCCESS if the call succeeds.

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

module→set_userData()**YModule****module→setUserData()**

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

js	function set_userData (data)
cpp	void set_userData (void* data)
m	-(void) setUserData : (id) data
pas	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)
php	function set_userData (\$data)
es	function 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()****YModule****module**→**setUserVar()**

Stores a 32 bit value in the device RAM.

js	function set_userVar (newval)
cpp	int set_userVar (int newval)
m	-(int) setUserVar : (int) newval
pas	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)
uwp	Task<int> set_userVar (int newval)
py	def set_userVar (newval)
php	function set_userVar (\$newval)
es	function set_userVar (newval)
cmd	YModule target set_userVar newval

This attribute is at programmer disposal, should he need to store a state variable. On startup and after a device reboot, the value is always reset to zero.

Parameters :

newval an integer

Returns :

YAPI_SUCCESS if the call succeeds.

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

module→triggerConfigChangeCallback()**YModule**

Triggers a configuration change callback, to check if they are supported or not.

js	function triggerConfigChangeCallback ()
cpp	int triggerConfigChangeCallback ()
m	-(int) triggerConfigChangeCallback
pas	function triggerConfigChangeCallback (): LongInt
vb	function triggerConfigChangeCallback () As Integer
cs	int triggerConfigChangeCallback ()
java	int triggerConfigChangeCallback ()
uwp	Task<int> triggerConfigChangeCallback ()
py	def triggerConfigChangeCallback ()
php	function triggerConfigChangeCallback ()
es	function triggerConfigChangeCallback ()
cmd	YModule target triggerConfigChangeCallback

module→triggerFirmwareUpdate()**YModule**

Schedules a module reboot into special firmware update mode.

js	function triggerFirmwareUpdate (secBeforeReboot)
cpp	int triggerFirmwareUpdate (int secBeforeReboot)
m	-(int) triggerFirmwareUpdate : (int) secBeforeReboot
pas	function triggerFirmwareUpdate (secBeforeReboot : LongInt): LongInt
vb	function triggerFirmwareUpdate () As Integer
cs	int triggerFirmwareUpdate (int secBeforeReboot)
java	int triggerFirmwareUpdate (int secBeforeReboot)
uwp	Task<int> triggerFirmwareUpdate (int secBeforeReboot)
py	def triggerFirmwareUpdate (secBeforeReboot)
php	function triggerFirmwareUpdate (\$ secBeforeReboot)
es	function triggerFirmwareUpdate (secBeforeReboot)
cmd	YModule target triggerFirmwareUpdate secBeforeReboot

Parameters :

secBeforeReboot number of seconds before rebooting

Returns :

YAPI_SUCCESS when the call succeeds.

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

module→updateFirmware()**YModule**

Prepares a firmware update of the module.

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

This method returns a YFirmwareUpdate object which handles the firmware update process.

Parameters :

path the path of the .byn file to use.

Returns :

a YFirmwareUpdate object or NULL on error.

module→updateFirmwareEx()**YModule**

Prepares a firmware update of the module.

js	function updateFirmwareEx (path , force)
cpp	YFirmwareUpdate updateFirmwareEx (string path , bool force)
m	-(YFirmwareUpdate*) updateFirmwareEx : (NSString*) path : (bool) force
pas	function updateFirmwareEx (path : string, force : boolean): TYFirmwareUpdate
vb	function updateFirmwareEx () As YFirmwareUpdate
cs	YFirmwareUpdate updateFirmwareEx (string path , bool force)
java	YFirmwareUpdate updateFirmwareEx (String path , boolean force)
uwp	Task<YFirmwareUpdate> updateFirmwareEx (string path , bool force)
py	def updateFirmwareEx (path , force)
php	function updateFirmwareEx (\$ path , \$ force)
es	function updateFirmwareEx (path , force)
cmd	YModule target updateFirmwareEx path force

This method returns a YFirmwareUpdate object which handles the firmware update process.

Parameters :

path the path of the .byn file to use.

force true to force the firmware update even if some prerequisites appear not to be met

Returns :

a YFirmwareUpdate object or NULL on error.

module→**wait_async()****YModule**

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

```
js function wait_async( callback, context)
```

```
es 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.

21.3. CarbonDioxide function interface

The Yoctopuce class YCarbonDioxide allows you to read and configure Yoctopuce CO2 sensors. It inherits from YSensor class the core functions to read measurements, to register callback functions, to access the autonomous datalogger. This class adds the ability to perform manual calibration if required.

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

es	in HTML: <code><script src="../../lib/yocto_carbondioxide.js"></script></code> in node.js: <code>require('yoctolib-es2017/yocto_carbondioxide.js');</code>
js	<code><script type='text/javascript' src='yocto_carbondioxide.js'></script></code>
cpp	<code>#include "yocto_carbondioxide.h"</code>
m	<code>#import "yocto_carbondioxide.h"</code>
pas	<code>uses yocto_carbondioxide;</code>
vb	<code>yocto_carbondioxide.vb</code>
cs	<code>yocto_carbondioxide.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YCarbonDioxide;</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YCarbonDioxide;</code>
py	<code>from yocto_carbondioxide import *</code>
php	<code>require_once('yocto_carbondioxide.php');</code>

Global functions

yFindCarbonDioxide(func)

Retrieves a CO2 sensor for a given identifier.

yFindCarbonDioxideInContext(yctx, func)

Retrieves a CO2 sensor for a given identifier in a YAPI context.

yFirstCarbonDioxide()

Starts the enumeration of CO2 sensors currently accessible.

yFirstCarbonDioxideInContext(yctx)

Starts the enumeration of CO2 sensors currently accessible.

YCarbonDioxide methods

carbondioxide→calibrateFromPoints(rawValues, refValues)

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

carbondioxide→clearCache()

Invalidates the cache.

carbondioxide→describe()

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

carbondioxide→get_abcPeriod()

Returns the Automatic Baseline Calibration period, in hours.

carbondioxide→get_advMode()

Returns the measuring mode used for the advertised value pushed to the parent hub.

carbondioxide→get_advertisedValue()

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

carbondioxide→get_currentRawValue()

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

carbondioxide→get_currentValue()

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

carbondioxide→get_dataLogger()

Returns the YDataLogger object of the device hosting the sensor.

carbondioxide→get_errorMessage()

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

carbondioxide→get_errorType()

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

carbondioxide→get_friendlyName()

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

carbondioxide→get_functionDescriptor()

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

carbondioxide→get_functionId()

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

carbondioxide→get_hardwareId()

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

carbondioxide→get_highestValue()

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

carbondioxide→get_logFrequency()

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

carbondioxide→get_logicalName()

Returns the logical name of the CO2 sensor.

carbondioxide→get_lowestValue()

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

carbondioxide→get_module()

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

carbondioxide→get_module_async(callback, context)

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

carbondioxide→get_recordedData(startTime, endTime)

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

carbondioxide→get_reportFrequency()

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

carbondioxide→get_resolution()

Returns the resolution of the measured values.

carbondioxide→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

carbondioxide→get_unit()

Returns the measuring unit for the CO2 concentration.

carbondioxide→get_userData()

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

carbondioxide→isOnline()

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

carbondioxide→isOnline_async(callback, context)

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

carbondioxide→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

carbondioxide→load(msValidity)

Preloads the CO2 sensor cache with a specified validity duration.

carbondioxide→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

carbondioxide→loadCalibrationPoints(rawValues, refValues)

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

carbondioxide→load_async(msValidity, callback, context)

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

carbondioxide→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

carbondioxide→nextCarbonDioxide()

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

carbondioxide→registerTimedReportCallback(callback)

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

carbondioxide→registerValueCallback(callback)

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

carbondioxide→set_abcPeriod(newval)

Changes Automatic Baseline Calibration period, in hours.

carbondioxide→set_advMode(newval)

Changes the measuring mode used for the advertised value pushed to the parent hub.

carbondioxide→set_highestValue(newval)

Changes the recorded maximal value observed.

carbondioxide→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

carbondioxide→set_logicalName(newval)

Changes the logical name of the CO2 sensor.

carbondioxide→set_lowestValue(newval)

Changes the recorded minimal value observed.

carbondioxide→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

carbondioxide→set_resolution(newval)

Changes the resolution of the measured physical values.

carbondioxide→set_userData(data)

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

carbondioxide→startDataLogger()

Starts the data logger on the device.

carbondioxide→stopDataLogger()

Stops the datalogger on the device.

carbondioxide→triggerBaselineCalibration()

Triggers a baseline calibration at standard CO2 ambient level (400ppm).

carbondioxide→triggerZeroCalibration()

Triggers a zero calibration of the sensor on carbon dioxide-free air.

carbondioxide→**unmuteValueCallbacks()**

Re-enables the propagation of every new advertised value to the parent hub.

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

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

YCarbonDioxide.FindCarbonDioxide() yFindCarbonDioxide()

YCarbonDioxide

Retrieves a CO2 sensor for a given identifier.

js	function yFindCarbonDioxide (func)
cpp	YCarbonDioxide* yFindCarbonDioxide (string func)
m	+(YCarbonDioxide*) FindCarbonDioxide : (NSString*) func
pas	function yFindCarbonDioxide (func : string): TYCarbonDioxide
vb	function yFindCarbonDioxide (ByVal func As String) As YCarbonDioxide
cs	YCarbonDioxide FindCarbonDioxide (string func)
java	YCarbonDioxide FindCarbonDioxide (String func)
uwp	YCarbonDioxide FindCarbonDioxide (string func)
py	def FindCarbonDioxide (func)
php	function yFindCarbonDioxide (\$func)
es	function FindCarbonDioxide (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 CO2 sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YCarbonDioxide.isOnline()` to test if the CO2 sensor is indeed online at a given time. In case of ambiguity when looking for a CO2 sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

If a call to this object's `is_online()` method returns `FALSE` although you are certain that the matching device is plugged, make sure that you did call `registerHub()` at application initialization time.

Parameters :

func a string that uniquely characterizes the CO2 sensor

Returns :

a `YCarbonDioxide` object allowing you to drive the CO2 sensor.

YCarbonDioxide.FindCarbonDioxideInContext() yFindCarbonDioxideInContext()

YCarbonDioxide

Retrieves a CO2 sensor for a given identifier in a YAPI context.

```
java YCarbonDioxide FindCarbonDioxideInContext( YAPIContext yctx,  
                                                String func)
```

```
uwp YCarbonDioxide FindCarbonDioxideInContext( YAPIContext yctx,  
                                                string func)
```

```
es function FindCarbonDioxideInContext( yctx, func)
```

The identifier can be specified using several formats:

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

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

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the CO2 sensor

Returns :

a `YCarbonDioxide` object allowing you to drive the CO2 sensor.

YCarbonDioxide.FirstCarbonDioxide() yFirstCarbonDioxide()

YCarbonDioxide

Starts the enumeration of CO2 sensors currently accessible.

js	function yFirstCarbonDioxide ()
cpp	YCarbonDioxide* yFirstCarbonDioxide ()
m	+(YCarbonDioxide*) FirstCarbonDioxide
pas	function yFirstCarbonDioxide (): TYCarbonDioxide
vb	function yFirstCarbonDioxide () As YCarbonDioxide
cs	YCarbonDioxide FirstCarbonDioxide ()
java	YCarbonDioxide FirstCarbonDioxide ()
uwp	YCarbonDioxide FirstCarbonDioxide ()
py	def FirstCarbonDioxide ()
php	function yFirstCarbonDioxide ()
es	function FirstCarbonDioxide ()

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

Returns :

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

YCarbonDioxide.FirstCarbonDioxideInContext() yFirstCarbonDioxideInContext()

YCarbonDioxide

Starts the enumeration of CO2 sensors currently accessible.

java	YCarbonDioxide FirstCarbonDioxideInContext(YAPIContext yctx)
uwp	YCarbonDioxide FirstCarbonDioxideInContext(YAPIContext yctx)
es	function FirstCarbonDioxideInContext(yctx)

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

Parameters :

yctx a YAPI context.

Returns :

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

carbondioxide→calibrateFromPoints()**YCarbonDioxide**

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

```

js function calibrateFromPoints( rawValues, refValues)
cpp int calibrateFromPoints( vector<double> rawValues,
                             vector<double> refValues)
m -(int) calibrateFromPoints : (NSMutableArray*) rawValues
   : (NSMutableArray*) refValues
pas 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)
uwp Task<int> calibrateFromPoints( List<double> rawValues,
                                   List<double> refValues)
py def calibrateFromPoints( rawValues, refValues)
php function calibrateFromPoints( $rawValues, $refValues)
es function calibrateFromPoints( rawValues, refValues)
cmd YCarbonDioxide target calibrateFromPoints rawValues refValues

```

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

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

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→clearCache()**YCarbonDioxide**

Invalidates the cache.

js	function clearCache ()
cpp	void clearCache ()
m	-(void) clearCache
pas	procedure clearCache ()
vb	procedure clearCache ()
cs	void clearCache ()
java	void clearCache ()
py	def clearCache ()
php	function clearCache ()
es	function clearCache ()

Invalidates the cache of the CO2 sensor attributes. Forces the next call to `get_xxx()` or `loadxxx()` to use values that come from the device.

carbondioxide→describe()**YCarbonDioxide**

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

js	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 ()
php	function describe ()
es	function describe ()

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=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 CO2 sensor (ex:
`Relay(MyCustomName.relay1)=RELAYL01-123456.relay1`)

carbondioxide→**get_abcPeriod()****YCarbonDioxide****carbondioxide**→**abcPeriod()**

Returns the Automatic Baseline Calibration period, in hours.

js	function get_abcPeriod ()
cpp	int get_abcPeriod ()
m	-(int) abcPeriod
pas	function get_abcPeriod (): LongInt
vb	function get_abcPeriod () As Integer
cs	int get_abcPeriod ()
java	int get_abcPeriod ()
uwp	Task<int> get_abcPeriod ()
py	def get_abcPeriod ()
php	function get_abcPeriod ()
es	function get_abcPeriod ()
cmd	YCarbonDioxide target get_abcPeriod

A negative value means that automatic baseline calibration is disabled.

Returns :

an integer corresponding to the Automatic Baseline Calibration period, in hours

On failure, throws an exception or returns Y_ABCPERIOD_INVALID.

carbondioxide→get_advMode()**YCarbonDioxide****carbondioxide→advMode()**

Returns the measuring mode used for the advertised value pushed to the parent hub.

js	function get_advMode ()
cpp	Y_ADVMODE_enum get_advMode ()
m	-(Y_ADVMODE_enum) advMode
pas	function get_advMode (): Integer
vb	function get_advMode () As Integer
cs	int get_advMode ()
java	int get_advMode ()
uwp	Task<int> get_advMode ()
py	def get_advMode ()
php	function get_advMode ()
es	function get_advMode ()
cmd	YCarbonDioxide target get_advMode

Returns :

a value among Y_ADVMODE_IMMEDIATE, Y_ADVMODE_PERIOD_AVG, Y_ADVMODE_PERIOD_MIN and Y_ADVMODE_PERIOD_MAX corresponding to the measuring mode used for the advertised value pushed to the parent hub

On failure, throws an exception or returns Y_ADVMODE_INVALID.

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

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

js	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 ()
uwp	Task<string> get_advertisedValue ()
py	def get_advertisedValue ()
php	function get_advertisedValue ()
es	function get_advertisedValue ()
cmd	YCarbonDioxide target get_advertisedValue

Returns :

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

On failure, throws an exception or returns Y_ADVERTISEDVALUE_INVALID.

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

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

js	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 ()
uwp	Task<double> get_currentRawValue ()
py	def get_currentRawValue ()
php	function get_currentRawValue ()
es	function get_currentRawValue ()
cmd	YCarbonDioxide target get_currentRawValue

Returns :

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

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

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

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

js	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 ()
uwp	Task<double> get_currentValue ()
py	def get_currentValue ()
php	function get_currentValue ()
es	function get_currentValue ()
cmd	YCarbonDioxide target get_currentValue

Returns :

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

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

carbondioxide→get_dataLogger()**YCarbonDioxide****carbondioxide→dataLogger()**

Returns the YDataLogger object of the device hosting the sensor.

js	function get_dataLogger ()
cpp	YDataLogger* get_dataLogger ()
m	-(YDataLogger*) dataLogger
pas	function get_dataLogger (): TYDataLogger
vb	function get_dataLogger () As YDataLogger
cs	YDataLogger get_dataLogger ()
java	YDataLogger get_dataLogger ()
uwp	Task<YDataLogger> get_dataLogger ()
py	def get_dataLogger ()
php	function get_dataLogger ()
es	function get_dataLogger ()

This method returns an object of class YDataLogger that can control global parameters of the data logger. The returned object should not be freed.

Returns :

an YDataLogger object or null on error.

carbondioxide→get_errorMessage()**YCarbonDioxide****carbondioxide→errorMessage()**

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

js	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 ()
php	function get_errorMessage ()
es	function 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 CO2 sensor object

carbondioxide→get_errorType()**YCarbonDioxide****carbondioxide→errorType()**

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

js	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 ()
php	function get_errorType ()
es	function 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 CO2 sensor object

carbondioxide→get_friendlyName()**YCarbonDioxide****carbondioxide→friendlyName()**

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

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

The returned string uses the logical names of the module and of the CO2 sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the CO2 sensor (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the CO2 sensor using logical names (ex: `MyCustomName.relay1`)

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

carbondioxide→get_functionDescriptor()**YCarbonDioxide****carbondioxide→functionDescriptor()**

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

js	function get_functionDescriptor ()
cpp	YFUN_DESCR get_functionDescriptor ()
m	-(YFUN_DESCR) functionDescriptor
pas	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 ()
php	function get_functionDescriptor ()
es	function 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.

carbondioxide→get_functionId()**YCarbonDioxide****carbondioxide→functionId()**

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

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

For example `relay1`

Returns :

a string that identifies the CO2 sensor (ex: `relay1`)

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

carbondioxide→get_hardwareId()**YCarbonDioxide****carbondioxide→hardwareId()**

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

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

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the CO2 sensor (for example RELAYL01-123456 . r e l a y 1).

Returns :

a string that uniquely identifies the CO2 sensor (ex: RELAYL01-123456 . r e l a y 1)

On failure, throws an exception or returns Y_HARDWAREID_INVALID.

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

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

js	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 ()
uwp	Task<double> get_highestValue ()
py	def get_highestValue ()
php	function get_highestValue ()
es	function get_highestValue ()
cmd	YCarbonDioxide target get_highestValue

Can be reset to an arbitrary value thanks to `set_highestValue()`.

Returns :

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

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

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

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

js	function get_logFrequency ()
cpp	string get_logFrequency ()
m	-(NSString*) logFrequency
pas	function get_logFrequency (): string
vb	function get_logFrequency () As String
cs	string get_logFrequency ()
java	String get_logFrequency ()
uwp	Task<string> get_logFrequency ()
py	def get_logFrequency ()
php	function get_logFrequency ()
es	function get_logFrequency ()
cmd	YCarbonDioxide target get_logFrequency

Returns :

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

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

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

Returns the logical name of the CO2 sensor.

js	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 ()
uwp	Task<string> get_logicalName ()
py	def get_logicalName ()
php	function get_logicalName ()
es	function get_logicalName ()
cmd	YCarbonDioxide target get_logicalName

Returns :

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

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

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

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

js	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 ()
uwp	Task<double> get_lowestValue ()
py	def get_lowestValue ()
php	function get_lowestValue ()
es	function get_lowestValue ()
cmd	YCarbonDioxide target get_lowestValue

Can be reset to an arbitrary value thanks to `set_lowestValue()`.

Returns :

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

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

carbondioxide→get_module()**YCarbonDioxide****carbondioxide→module()**

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

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

carbondioxide→get_module_async()**YCarbonDioxide****carbondioxide→module_async()**

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

```
js function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as on-line.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous 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.

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

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

js	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)
uwp	Task<YDataSet> get_recordedData (long startTime , long endTime)
py	def get_recordedData (startTime , endTime)
php	function get_recordedData (\$ startTime , \$ endTime)
es	function get_recordedData (startTime , endTime)
cmd	YCarbonDioxide target get_recordedData startTime endTime

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

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

Parameters :

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

Returns :

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

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

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

js	function get_reportFrequency ()
cpp	string get_reportFrequency ()
m	-(NSString*) reportFrequency
pas	function get_reportFrequency (): string
vb	function get_reportFrequency () As String
cs	string get_reportFrequency ()
java	String get_reportFrequency ()
uwp	Task<string> get_reportFrequency ()
py	def get_reportFrequency ()
php	function get_reportFrequency ()
es	function get_reportFrequency ()
cmd	YCarbonDioxide target get_reportFrequency

Returns :

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

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

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

Returns the resolution of the measured values.

js	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 ()
uwp	Task<double> get_resolution ()
py	def get_resolution ()
php	function get_resolution ()
es	function get_resolution ()
cmd	YCarbonDioxide target get_resolution

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

Returns :

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

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

carbondioxide→get_sensorState()**YCarbonDioxide****carbondioxide→sensorState()**

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

js	function get_sensorState ()
cpp	int get_sensorState ()
m	-(int) sensorState
pas	function get_sensorState (): LongInt
vb	function get_sensorState () As Integer
cs	int get_sensorState ()
java	int get_sensorState ()
uwp	Task<int> get_sensorState ()
py	def get_sensorState ()
php	function get_sensorState ()
es	function get_sensorState ()
cmd	YCarbonDioxide target get_sensorState

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns Y_SENSORSTATE_INVALID.

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

Returns the measuring unit for the CO2 concentration.

js	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 ()
uwp	Task<string> get_unit ()
py	def get_unit ()
php	function get_unit ()
es	function get_unit ()
cmd	YCarbonDioxide target get_unit

Returns :

a string corresponding to the measuring unit for the CO2 concentration

On failure, throws an exception or returns Y_UNIT_INVALID.

carbondioxide→get_userData()**YCarbonDioxide****carbondioxide→userData()**

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

js	function get_userData ()
cpp	void * get_userData ()
m	-(id) userData
pas	function get_userData (): Tobject
vb	function get_userData () As Object
cs	object get_userData ()
java	Object get_userData ()
py	def get_userData ()
php	function get_userData ()
es	function 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.

carbondioxide→isOnline()**YCarbonDioxide**

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

js	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 ()
php	function isOnline ()
es	function isOnline ()

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

Returns :

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

carbondioxide→isOnline_async()**YCarbonDioxide**

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

```
js function isOnline_async( callback, context)
```

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

carbondioxide→isSensorReady()**YCarbonDioxide**

Checks if the sensor is currently able to provide an up-to-date measure.

```
cmd YCarbonDioxide target isSensorReady
```

Returns false if the device is unreachable, or if the sensor does not have a current measure to transmit. No exception is raised if there is an error while trying to contact the device hosting \$THEFUNCTION\$.

Returns :

`true` if the sensor can provide an up-to-date measure, and `false` otherwise

carbondioxide→load()**YCarbonDioxide**

Preloads the CO2 sensor cache with a specified validity duration.

js	function load (msValidity)
cpp	YRETCODE load (int msValidity)
m	-(YRETCODE) load : (u64) msValidity
pas	function load (msValidity : u64): YRETCODE
vb	function load (ByVal msValidity As Long) As YRETCODE
cs	YRETCODE load (ulong msValidity)
java	int load (long msValidity)
py	def load (msValidity)
php	function load (\$msValidity)
es	function load (msValidity)

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

Parameters :

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

Returns :

YAPI_SUCCESS when the call succeeds.

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

carbondioxide→loadAttribute()**YCarbonDioxide**

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

js	function loadAttribute (attrName)
cpp	string loadAttribute (string attrName)
m	-(NSString*) loadAttribute : (NSString*) attrName
pas	function loadAttribute (attrName : string): string
vb	function loadAttribute () As String
cs	string loadAttribute (string attrName)
java	String loadAttribute (String attrName)
uwp	Task<string> loadAttribute (string attrName)
py	def loadAttribute (attrName)
php	function loadAttribute (\$ attrName)
es	function loadAttribute (attrName)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

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

carbondioxide→loadCalibrationPoints()**YCarbonDioxide**

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

```

js function loadCalibrationPoints( rawValues, refValues)
cpp int loadCalibrationPoints( vector<double>& rawValues,
                             vector<double>& refValues)
m  -(int) loadCalibrationPoints : (NSMutableArray*) rawValues
                             : (NSMutableArray*) refValues
pas 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)
uwp Task<int> loadCalibrationPoints( List<double> rawValues,
                                   List<double> refValues)
py  def loadCalibrationPoints( rawValues, refValues)
php function loadCalibrationPoints( &$rawValues, &$refValues)
es  function loadCalibrationPoints( rawValues, refValues)
cmd YCarbonDioxide target loadCalibrationPoints rawValues refValues

```

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→load_async()**YCarbonDioxide**

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

```
js function load_async( msValidity, callback, context)
```

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

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

- msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds
- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or YAPI_SUCCESS)
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

carbondioxide→muteValueCallbacks()**YCarbonDioxide**

Disables the propagation of every new advertised value to the parent hub.

js	function muteValueCallbacks ()
cpp	int muteValueCallbacks ()
m	-(int) muteValueCallbacks
pas	function muteValueCallbacks (): LongInt
vb	function muteValueCallbacks () As Integer
cs	int muteValueCallbacks ()
java	int muteValueCallbacks ()
uwp	Task<int> muteValueCallbacks ()
py	def muteValueCallbacks ()
php	function muteValueCallbacks ()
es	function muteValueCallbacks ()
cmd	YCarbonDioxide target muteValueCallbacks

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

YAPI_SUCCESS when the call succeeds.

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

carbondioxide→nextCarbonDioxide()**YCarbonDioxide**

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

js	function nextCarbonDioxide ()
cpp	YCarbonDioxide * nextCarbonDioxide ()
m	-(YCarbonDioxide*) nextCarbonDioxide
pas	function nextCarbonDioxide (): TYCarbonDioxide
vb	function nextCarbonDioxide () As YCarbonDioxide
cs	YCarbonDioxide nextCarbonDioxide ()
java	YCarbonDioxide nextCarbonDioxide ()
uwp	YCarbonDioxide nextCarbonDioxide ()
py	def nextCarbonDioxide ()
php	function nextCarbonDioxide ()
es	function nextCarbonDioxide ()

Returns :

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

carbondioxide→registerTimedReportCallback()**YCarbonDioxide**

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

js	function registerTimedReportCallback (callback)
cpp	int registerTimedReportCallback (YCarbonDioxideTimedReportCallback callback)
m	-(int) registerTimedReportCallback : (YCarbonDioxideTimedReportCallback) callback
pas	function registerTimedReportCallback (callback : TYCarbonDioxideTimedReportCallback): LongInt
vb	function registerTimedReportCallback () As Integer
cs	int registerTimedReportCallback (TimedReportCallback callback)
java	int registerTimedReportCallback (TimedReportCallback callback)
uwp	Task<int> registerTimedReportCallback (TimedReportCallback callback)
py	def registerTimedReportCallback (callback)
php	function registerTimedReportCallback (\$callback)
es	function 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.

carbondioxide→registerValueCallback()**YCarbonDioxide**

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

js	function registerValueCallback (callback)
cpp	int registerValueCallback (YCarbonDioxideValueCallback callback)
m	-(int) registerValueCallback : (YCarbonDioxideValueCallback) callback
pas	function registerValueCallback (callback : TYCarbonDioxideValueCallback): LongInt
vb	function registerValueCallback () As Integer
cs	int registerValueCallback (ValueCallback callback)
java	int registerValueCallback (UpdateCallback callback)
uwp	Task<int> registerValueCallback (ValueCallback callback)
py	def registerValueCallback (callback)
php	function registerValueCallback (\$callback)
es	function 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.

carbondioxide→set_abcPeriod()**YCarbonDioxide****carbondioxide→setAbcPeriod()**

Changes Automatic Baseline Calibration period, in hours.

js	function set_abcPeriod (newval)
cpp	int set_abcPeriod (int newval)
m	-(int) setAbcPeriod : (int) newval
pas	function set_abcPeriod (newval : LongInt): integer
vb	function set_abcPeriod (ByVal newval As Integer) As Integer
cs	int set_abcPeriod (int newval)
java	int set_abcPeriod (int newval)
uwp	Task<int> set_abcPeriod (int newval)
py	def set_abcPeriod (newval)
php	function set_abcPeriod (\$newval)
es	function set_abcPeriod (newval)
cmd	YCarbonDioxide target set_abcPeriod newval

If you need to disable automatic baseline calibration (for instance when using the sensor in an environment that is constantly above 400ppm CO2), set the period to -1. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer corresponding to Automatic Baseline Calibration period, in hours

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_advMode()**YCarbonDioxide****carbondioxide→setAdvMode()**

Changes the measuring mode used for the advertised value pushed to the parent hub.

js	function set_advMode (newval)
cpp	int set_advMode (Y_ADVMODE_enum newval)
m	-(int) setAdvMode : (Y_ADVMODE_enum) newval
pas	function set_advMode (newval : Integer): integer
vb	function set_advMode (ByVal newval As Integer) As Integer
cs	int set_advMode (int newval)
java	int set_advMode (int newval)
uwp	Task<int> set_advMode (int newval)
py	def set_advMode (newval)
php	function set_advMode (\$newval)
es	function set_advMode (newval)
cmd	YCarbonDioxide target set_advMode newval

Parameters :

newval a value among Y_ADVMODE_IMMEDIATE, Y_ADVMODE_PERIOD_AVG, Y_ADVMODE_PERIOD_MIN and Y_ADVMODE_PERIOD_MAX corresponding to the measuring mode used for the advertised value pushed to the parent hub

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the recorded maximal value observed.

js	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)
uwp	Task<int> set_highestValue (double newval)
py	def set_highestValue (newval)
php	function set_highestValue (\$newval)
es	function set_highestValue (newval)
cmd	YCarbonDioxide target set_highestValue newval

Can be used to reset the value returned by get_lowestValue().

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_logFrequency() carbondioxide→setLogFrequency()

YCarbonDioxide

Changes the datalogger recording frequency for this function.

js	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)
uwp	Task<int> set_logFrequency (string newval)
py	def set_logFrequency (newval)
php	function set_logFrequency (\$ newval)
es	function set_logFrequency (newval)
cmd	YCarbonDioxide target set_logFrequency newval

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the logical name of the CO2 sensor.

js	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)
uwp	Task<int> set_logicalName (string newval)
py	def set_logicalName (newval)
php	function set_logicalName (\$ newval)
es	function set_logicalName (newval)
cmd	YCarbonDioxide target set_logicalName newval

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the recorded minimal value observed.

js	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)
uwp	Task<int> set_lowestValue (double newval)
py	def set_lowestValue (newval)
php	function set_lowestValue (\$newval)
es	function set_lowestValue (newval)
cmd	YCarbonDioxide target set_lowestValue newval

Can be used to reset the value returned by get_lowestValue().

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the timed value notification frequency for this function.

js	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)
uwp	Task<int> set_reportFrequency (string newval)
py	def set_reportFrequency (newval)
php	function set_reportFrequency (\$ newval)
es	function set_reportFrequency (newval)
cmd	YCarbonDioxide target set_reportFrequency newval

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_resolution() carbondioxide→setResolution()

YCarbonDioxide

Changes the resolution of the measured physical values.

js	function set_resolution (newval)
cpp	int set_resolution (double newval)
m	-(int) setResolution : (double) newval
pas	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)
uwp	Task<int> set_resolution (double newval)
py	def set_resolution (newval)
php	function set_resolution (\$newval)
es	function set_resolution (newval)
cmd	YCarbonDioxide target set_resolution newval

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→set_userData()**YCarbonDioxide****carbondioxide→setUserData()**

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

js	function set_userData (data)
cpp	void set_userData (void* data)
m	-(void) setUserData : (id) data
pas	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)
php	function set_userData (\$data)
es	function 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

carbondioxide→startDataLogger()**YCarbonDioxide**

Starts the data logger on the device.

js	function startDataLogger ()
cpp	int startDataLogger ()
m	-(int) startDataLogger
pas	function startDataLogger (): LongInt
vb	function startDataLogger () As Integer
cs	int startDataLogger ()
java	int startDataLogger ()
uwp	Task<int> startDataLogger ()
py	def startDataLogger ()
php	function startDataLogger ()
es	function startDataLogger ()
cmd	YCarbonDioxide target startDataLogger

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :

YAPI_SUCCESS if the call succeeds.

carbondioxide→stopDataLogger()**YCarbonDioxide**

Stops the datalogger on the device.

js	function stopDataLogger ()
cpp	int stopDataLogger ()
m	-(int) stopDataLogger
pas	function stopDataLogger (): LongInt
vb	function stopDataLogger () As Integer
cs	int stopDataLogger ()
java	int stopDataLogger ()
uwp	Task<int> stopDataLogger ()
py	def stopDataLogger ()
php	function stopDataLogger ()
es	function stopDataLogger ()
cmd	YCarbonDioxide target stopDataLogger

Returns :

YAPI_SUCCESS if the call succeeds.

carbondioxide→triggerBaselineCalibration()**YCarbonDioxide**

Triggers a baseline calibration at standard CO2 ambient level (400ppm).

js	function triggerBaselineCalibration ()
cpp	int triggerBaselineCalibration ()
m	-(int) triggerBaselineCalibration
pas	function triggerBaselineCalibration (): LongInt
vb	function triggerBaselineCalibration () As Integer
cs	int triggerBaselineCalibration ()
java	int triggerBaselineCalibration ()
uwp	Task<int> triggerBaselineCalibration ()
py	def triggerBaselineCalibration ()
php	function triggerBaselineCalibration ()
es	function triggerBaselineCalibration ()
cmd	YCarbonDioxide target triggerBaselineCalibration

It is normally not necessary to manually calibrate the sensor, because the built-in automatic baseline calibration procedure will automatically fix any long-term drift based on the lowest level of CO2 observed over the automatic calibration period. However, if you disable automatic baseline calibration, you may want to manually trigger a calibration from time to time. Before starting a baseline calibration, make sure to put the sensor in a standard environment (e.g. outside in fresh air) at around 400ppm.

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→triggerZeroCalibration()**YCarbonDioxide**

Triggers a zero calibration of the sensor on carbon dioxide-free air.

js	function triggerZeroCalibration ()
cpp	int triggerZeroCalibration ()
m	-(int) triggerZeroCalibration
pas	function triggerZeroCalibration (): LongInt
vb	function triggerZeroCalibration () As Integer
cs	int triggerZeroCalibration ()
java	int triggerZeroCalibration ()
uwp	Task<int> triggerZeroCalibration ()
py	def triggerZeroCalibration ()
php	function triggerZeroCalibration ()
es	function triggerZeroCalibration ()
cmd	YCarbonDioxide target triggerZeroCalibration

It is normally not necessary to manually calibrate the sensor, because the built-in automatic baseline calibration procedure will automatically fix any long-term drift based on the lowest level of CO2 observed over the automatic calibration period. However, if you disable automatic baseline calibration, you may want to manually trigger a calibration from time to time. Before starting a zero calibration, you should circulate carbon dioxide-free air within the sensor for a minute or two, using a small pipe connected to the sensor. Please contact support@yoctopuce.com for more details on the zero calibration procedure.

Returns :

YAPI_SUCCESS if the call succeeds.

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

carbondioxide→unmuteValueCallbacks()**YCarbonDioxide**

Re-enables the propagation of every new advertised value to the parent hub.

js	function unmuteValueCallbacks ()
cpp	int unmuteValueCallbacks ()
m	-(int) unmuteValueCallbacks
pas	function unmuteValueCallbacks (): LongInt
vb	function unmuteValueCallbacks () As Integer
cs	int unmuteValueCallbacks ()
java	int unmuteValueCallbacks ()
uwp	Task<int> unmuteValueCallbacks ()
py	def unmuteValueCallbacks ()
php	function unmuteValueCallbacks ()
es	function unmuteValueCallbacks ()
cmd	YCarbonDioxide target unmuteValueCallbacks

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

YAPI_SUCCESS when the call succeeds.

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

carbondioxide→wait_async()**YCarbonDioxide**

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

```
js function wait_async( callback, context)
```

```
es 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.

21.4. 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	<code><script type='text/javascript' src='yocto_api.js'></script></code>
cpp	<code>#include "yocto_api.h"</code>
m	<code>#import "yocto_api.h"</code>
pas	<code>uses yocto_api;</code>
vb	<code>yocto_api.vb</code>
cs	<code>yocto_api.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YModule;</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YModule;</code>
py	<code>from yocto_api import *</code>
php	<code>require_once('yocto_api.php');</code>
es	in HTML: <code><script src='../lib/yocto_api.js'></script></code> in node.js: <code>require('yoctolib-es2017/yocto_api.js');</code>

Global functions

yFindDataLogger(func)

Retrieves a data logger for a given identifier.

yFindDataLoggerInContext(yctx, func)

Retrieves a data logger for a given identifier in a YAPI context.

yFirstDataLogger()

Starts the enumeration of data loggers currently accessible.

yFirstDataLoggerInContext(yctx)

Starts the enumeration of data loggers currently accessible.

YDataLogger methods

datalogger→clearCache()

Invalidates the cache.

datalogger→describe()

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

datalogger→forgetAllDataStreams()

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

datalogger→get_advertisedValue()

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

datalogger→get_autoStart()

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

datalogger→get_beaconDriven()

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

datalogger→get_currentRunIndex()

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

datalogger→get_dataSets()

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

datalogger→get_dataStreams(v)

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

datalogger→get_errorMessage()

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

datalogger→get_errorType()

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

datalogger→get_friendlyName()

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

datalogger→get_functionDescriptor()

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

datalogger→get_functionId()

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

datalogger→get_hardwareId()

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

datalogger→get_logicalName()

Returns the logical name of the data logger.

datalogger→get_module()

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

datalogger→get_module_async(callback, context)

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

datalogger→get_recording()

Returns the current activation state of the data logger.

datalogger→get_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→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

datalogger→load_async(msValidity, callback, context)

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

datalogger→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

datalogger→nextDataLogger()

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

datalogger→registerValueCallback(callback)

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

datalogger→set_autoStart(newval)

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

datalogger→set_beaconDriven(newval)

Changes the type of synchronisation of the data logger.

datalogger→set_logicalName(newval)

Changes the logical name of the data logger.

datalogger→set_recording(newval)

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

datalogger→set_timeUTC(newval)

Changes the current UTC time reference used for recorded data.

datalogger→set_userData(data)

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

datalogger→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

datalogger→wait_async(callback, context)

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

YDataLogger.FindDataLogger() yFindDataLogger()

YDataLogger

Retrieves a data logger for a given identifier.

js	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)
uwp	YDataLogger FindDataLogger (string func)
py	def FindDataLogger (func)
php	function yFindDataLogger (\$func)
es	function FindDataLogger (func)

The identifier can be specified using several formats:

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

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

If a call to this object's `is_online()` method returns `FALSE` although you are certain that the matching device is plugged, make sure that you did call `registerHub()` at application initialization time.

Parameters :

func a string that uniquely characterizes the data logger

Returns :

a `YDataLogger` object allowing you to drive the data logger.

YDataLogger.FindDataLoggerInContext() yFindDataLoggerInContext()

YDataLogger

Retrieves a data logger for a given identifier in a YAPI context.

```
java YDataLogger FindDataLoggerInContext( YAPIContext yctx,  
                                           String func)
```

```
uwp YDataLogger FindDataLoggerInContext( YAPIContext yctx,  
                                           string func)
```

```
es function FindDataLoggerInContext( yctx, func)
```

The identifier can be specified using several formats:

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

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

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the data logger

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 ()
cpp	YDataLogger* yFirstDataLogger ()
m	+(YDataLogger*) FirstDataLogger
pas	function yFirstDataLogger (): TYDataLogger
vb	function yFirstDataLogger () As YDataLogger
cs	YDataLogger FirstDataLogger ()
java	YDataLogger FirstDataLogger ()
uwp	YDataLogger FirstDataLogger ()
py	def FirstDataLogger ()
php	function yFirstDataLogger ()
es	function FirstDataLogger ()

Use the method `YDataLogger.nextDataLogger( )` to iterate on next data loggers.

Returns :

a pointer to a `YDataLogger` object, corresponding to the first data logger currently online, or a `null` pointer if there are none.

YDataLogger.FirstDataLoggerInContext() yFirstDataLoggerInContext()

YDataLogger

Starts the enumeration of data loggers currently accessible.

```
java YDataLogger FirstDataLoggerInContext( YAPIContext yctx)
uwp YDataLogger FirstDataLoggerInContext( YAPIContext yctx)
es function FirstDataLoggerInContext( yctx)
```

Use the method YDataLogger .nextDataLogger () to iterate on next data loggers.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a YDataLogger object, corresponding to the first data logger currently online, or a null pointer if there are none.

datalogger→clearCache()**YDataLogger**

Invalidates the cache.

js	function clearCache ()
cpp	void clearCache ()
m	-(void) clearCache
pas	procedure clearCache ()
vb	procedure clearCache ()
cs	void clearCache ()
java	void clearCache ()
py	def clearCache ()
php	function clearCache ()
es	function clearCache ()

Invalidates the cache of the data logger attributes. Forces the next call to `get_xxx()` or `loadxxx()` to use values that come from the device.

datalogger→describe()**YDataLogger**

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

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

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=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.

js	function forgetAllDataStreams ()
cpp	int forgetAllDataStreams ()
m	-(int) forgetAllDataStreams
pas	function forgetAllDataStreams (): LongInt
vb	function forgetAllDataStreams () As Integer
cs	int forgetAllDataStreams ()
java	int forgetAllDataStreams ()
uwp	Task<int> forgetAllDataStreams ()
py	def forgetAllDataStreams ()
php	function forgetAllDataStreams ()
es	function forgetAllDataStreams ()
cmd	YDataLogger target forgetAllDataStreams

This method also resets the current run index to zero.

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

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

js	function get_advertisedValue ()
cpp	string get_advertisedValue ()
m	-(NSString*) advertisedValue
pas	function get_advertisedValue (): string
vb	function get_advertisedValue () As String
cs	string get_advertisedValue ()
java	String get_advertisedValue ()
uwp	Task<string> get_advertisedValue ()
py	def get_advertisedValue ()
php	function get_advertisedValue ()
es	function 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()**YDataLogger****datalogger→autoStart()**

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

js	function get_autoStart ()
cpp	Y_AUTOSTART_enum get_autoStart ()
m	-(Y_AUTOSTART_enum) autoStart
pas	function get_autoStart (): Integer
vb	function get_autoStart () As Integer
cs	int get_autoStart ()
java	int get_autoStart ()
uwp	Task<int> get_autoStart ()
py	def get_autoStart ()
php	function get_autoStart ()
es	function 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()**YDataLogger****datalogger→beaconDriven()**

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

js	function get_beaconDriven ()
cpp	Y_BEACONDRIVEN_enum get_beaconDriven ()
m	-(Y_BEACONDRIVEN_enum) beaconDriven
pas	function get_beaconDriven (): Integer
vb	function get_beaconDriven () As Integer
cs	int get_beaconDriven ()
java	int get_beaconDriven ()
uwp	Task<int> get_beaconDriven ()
py	def get_beaconDriven ()
php	function get_beaconDriven ()
es	function get_beaconDriven ()
cmd	YDataLogger target get_beaconDriven

Returns :

either Y_BEACONDRIVEN_OFF or Y_BEACONDRIVEN_ON, according to true if the data logger is synchronised with the localization beacon

On failure, throws an exception or returns Y_BEACONDRIVEN_INVALID.

datalogger→get_currentRunIndex()**YDataLogger****datalogger→currentRunIndex()**

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

js	function get_currentRunIndex ()
cpp	int get_currentRunIndex ()
m	-(int) currentRunIndex
pas	function get_currentRunIndex (): LongInt
vb	function get_currentRunIndex () As Integer
cs	int get_currentRunIndex ()
java	int get_currentRunIndex ()
uwp	Task<int> get_currentRunIndex ()
py	def get_currentRunIndex ()
php	function get_currentRunIndex ()
es	function 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()**YDataLogger****datalogger→dataSets()**

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

js	function get_dataSets ()
cpp	vector<YDataSet> get_dataSets ()
m	-(NSMutableArray*) dataSets
pas	function get_dataSets (): TYDataSetArray
vb	function get_dataSets () As List
cs	List<YDataSet> get_dataSets ()
java	ArrayList<YDataSet> get_dataSets ()
uwp	Task<List<YDataSet>> get_dataSets ()
py	def get_dataSets ()
php	function get_dataSets ()
es	function get_dataSets ()
cmd	YDataLogger target get_dataSets

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

Returns :

a list of YDataSet object.

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

datalogger→get_dataStreams()**YDataLogger****datalogger→dataStreams()**

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

js	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)
php	function get_dataStreams (&\$v)
es	function 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()****YDataLogger****datalogger**→**errorMessage()**

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

js	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 ()
php	function get_errorMessage ()
es	function 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 data logger object

datalogger→get_errorType()**YDataLogger****datalogger→errorType()**

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

js	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 ()
php	function get_errorType ()
es	function 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()**YDataLogger****datalogger→friendlyName()**

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

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

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

Returns :

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

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

datalogger→get_functionDescriptor()**YDataLogger****datalogger→functionDescriptor()**

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

js	function get_functionDescriptor ()
cpp	YFUN_DESCR get_functionDescriptor ()
m	-(YFUN_DESCR) functionDescriptor
pas	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 ()
php	function get_functionDescriptor ()
es	function 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()****YDataLogger****datalogger**→**functionId()**

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

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

For example `relay1`

Returns :

a string that identifies the data logger (ex: `relay1`)

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

datalogger→get_hardwareId()**YDataLogger****datalogger→hardwareId()**

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

js	function get_hardwareId ()
cpp	string get_hardwareId ()
m	-(NSString*) hardwareId
vb	function get_hardwareId () As String
cs	string get_hardwareId ()
java	String get_hardwareId ()
py	def get_hardwareId ()
php	function get_hardwareId ()
es	function 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()****YDataLogger****datalogger**→**logicalName()**

Returns the logical name of the data logger.

js	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 ()
uwp	Task<string> get_logicalName ()
py	def get_logicalName ()
php	function get_logicalName ()
es	function 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()**YDataLogger****datalogger→module()**

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

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

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

Returns :

an instance of YModule

datalogger→get_module_async()**YDataLogger****datalogger→module_async()**

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

```
js function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as on-line.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous 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()**YDataLogger****datalogger→recording()**

Returns the current activation state of the data logger.

js	function get_recording ()
cpp	Y_RECORDING_enum get_recording ()
m	-(Y_RECORDING_enum) recording
pas	function get_recording (): Integer
vb	function get_recording () As Integer
cs	int get_recording ()
java	int get_recording ()
uwp	Task<int> get_recording ()
py	def get_recording ()
php	function get_recording ()
es	function get_recording ()
cmd	YDataLogger target get_recording

Returns :

a value among Y_RECORDING_OFF, Y_RECORDING_ON and Y_RECORDING_PENDING corresponding to the current activation state of the data logger

On failure, throws an exception or returns Y_RECORDING_INVALID.

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

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

js	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 ()
uwp	Task<long> get_timeUTC ()
py	def get_timeUTC ()
php	function get_timeUTC ()
es	function 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()**YDataLogger****datalogger→userData()**

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

js	function get_userData ()
cpp	void * get_userData ()
m	-(id) userData
pas	function get_userData (): Tobject
vb	function get_userData () As Object
cs	object get_userData ()
java	Object get_userData ()
py	def get_userData ()
php	function get_userData ()
es	function get_userData ()

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

Returns :

the object stored previously by the caller.

datalogger→isOnline()**YDataLogger**

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

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

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

Returns :

`true` if the data logger can be reached, and `false` otherwise

datalogger→isOnline_async()**YDataLogger**

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

```
js function isOnline_async( callback, context)
```

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

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

datalogger→load()**YDataLogger**

Preloads the data logger cache with a specified validity duration.

js	function load (msValidity)
cpp	YRETCODE load (int msValidity)
m	-(YRETCODE) load : (u64) msValidity
pas	function load (msValidity : u64): YRETCODE
vb	function load (ByVal msValidity As Long) As YRETCODE
cs	YRETCODE load (ulong msValidity)
java	int load (long msValidity)
py	def load (msValidity)
php	function load (\$msValidity)
es	function load (msValidity)

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

Parameters :

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

Returns :

YAPI_SUCCESS when the call succeeds.

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

datalogger→loadAttribute()**YDataLogger**

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

js	function loadAttribute (attrName)
cpp	string loadAttribute (string attrName)
m	-(NSString*) loadAttribute : (NSString*) attrName
pas	function loadAttribute (attrName : string): string
vb	function loadAttribute () As String
cs	string loadAttribute (string attrName)
java	String loadAttribute (String attrName)
uwp	Task<string> loadAttribute (string attrName)
py	def loadAttribute (attrName)
php	function loadAttribute (\$attrName)
es	function loadAttribute (attrName)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

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

datalogger→load_async()**YDataLogger**

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

```
js function load_async( msValidity, callback, context)
```

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

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

- msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds
- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or YAPI_SUCCESS)
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

datalogger→muteValueCallbacks()**YDataLogger**

Disables the propagation of every new advertised value to the parent hub.

js	function muteValueCallbacks ()
cpp	int muteValueCallbacks ()
m	-(int) muteValueCallbacks
pas	function muteValueCallbacks (): LongInt
vb	function muteValueCallbacks () As Integer
cs	int muteValueCallbacks ()
java	int muteValueCallbacks ()
uwp	Task<int> muteValueCallbacks ()
py	def muteValueCallbacks ()
php	function muteValueCallbacks ()
es	function muteValueCallbacks ()
cmd	YDataLogger target muteValueCallbacks

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

YAPI_SUCCESS when the call succeeds.

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

datalogger→**nextDataLogger()****YDataLogger**

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

js	function nextDataLogger ()
cpp	YDataLogger * nextDataLogger ()
m	-(YDataLogger*) nextDataLogger
pas	function nextDataLogger (): TYDataLogger
vb	function nextDataLogger () As YDataLogger
cs	YDataLogger nextDataLogger ()
java	YDataLogger nextDataLogger ()
uwp	YDataLogger nextDataLogger ()
py	def nextDataLogger ()
php	function nextDataLogger ()
es	function nextDataLogger ()

Returns :

a pointer to a `YDataLogger` object, corresponding to a data logger currently online, or a `null` pointer if there are no more data loggers to enumerate.

datalogger→registerValueCallback()**YDataLogger**

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

js	function registerValueCallback (callback)
cpp	int registerValueCallback (YDataLoggerValueCallback callback)
m	-(int) registerValueCallback : (YDataLoggerValueCallback) callback
pas	function registerValueCallback (callback : TYDataLoggerValueCallback): LongInt
vb	function registerValueCallback () As Integer
cs	int registerValueCallback (ValueCallback callback)
java	int registerValueCallback (UpdateCallback callback)
uwp	Task<int> registerValueCallback (ValueCallback callback)
py	def registerValueCallback (callback)
php	function registerValueCallback (\$callback)
es	function registerValueCallback (callback)

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

Parameters :

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

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

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

js	function set_autoStart (newval)
cpp	int set_autoStart (Y_AUTOSTART_enum newval)
m	-(int) setAutoStart : (Y_AUTOSTART_enum) newval
pas	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)
uwp	Task<int> set_autoStart (int newval)
py	def set_autoStart (newval)
php	function set_autoStart (\$newval)
es	function 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()**YDataLogger****datalogger→setBeaconDriven()**

Changes the type of synchronisation of the data logger.

js	function set_beaconDriven (newval)
cpp	int set_beaconDriven (Y_BEACONDRIVEN_enum newval)
m	-(int) setBeaconDriven : (Y_BEACONDRIVEN_enum) newval
pas	function set_beaconDriven (newval : Integer): integer
vb	function set_beaconDriven (ByVal newval As Integer) As Integer
cs	int set_beaconDriven (int newval)
java	int set_beaconDriven (int newval)
uwp	Task<int> set_beaconDriven (int newval)
py	def set_beaconDriven (newval)
php	function set_beaconDriven (\$newval)
es	function set_beaconDriven (newval)
cmd	YDataLogger target set_beaconDriven newval

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the logical name of the data logger.

js	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)
uwp	Task<int> set_logicalName (string newval)
py	def set_logicalName (newval)
php	function set_logicalName (\$newval)
es	function 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()**YDataLogger****datalogger→setRecording()**

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

js	function set_recording (newval)
cpp	int set_recording (Y_RECORDING_enum newval)
m	-(int) setRecording : (Y_RECORDING_enum) newval
pas	function set_recording (newval : Integer): integer
vb	function set_recording (ByVal newval As Integer) As Integer
cs	int set_recording (int newval)
java	int set_recording (int newval)
uwp	Task<int> set_recording (int newval)
py	def set_recording (newval)
php	function set_recording (\$newval)
es	function set_recording (newval)
cmd	YDataLogger target set_recording newval

Parameters :

newval a value among Y_RECORDING_OFF, Y_RECORDING_ON and Y_RECORDING_PENDING corresponding to the activation state of the data logger to start/stop recording data

Returns :

YAPI_SUCCESS if the call succeeds.

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

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

Changes the current UTC time reference used for recorded data.

js	function set_timeUTC (newval)
cpp	int set_timeUTC (s64 newval)
m	-(int) setTimeUTC : (s64) newval
pas	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)
uwp	Task<int> set_timeUTC (long newval)
py	def set_timeUTC (newval)
php	function set_timeUTC (\$newval)
es	function 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()**YDataLogger****datalogger→setUserData()**

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

js	function set_userdata (data)
cpp	void set_userdata (void* data)
m	-(void) setUserData : (id) data
pas	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)
php	function set_userdata (\$data)
es	function set_userdata (data)

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

Parameters :

data any kind of object to be stored

datalogger→unmuteValueCallbacks()**YDataLogger**

Re-enables the propagation of every new advertised value to the parent hub.

js	function unmuteValueCallbacks ()
cpp	int unmuteValueCallbacks ()
m	-(int) unmuteValueCallbacks
pas	function unmuteValueCallbacks (): LongInt
vb	function unmuteValueCallbacks () As Integer
cs	int unmuteValueCallbacks ()
java	int unmuteValueCallbacks ()
uwp	Task<int> unmuteValueCallbacks ()
py	def unmuteValueCallbacks ()
php	function unmuteValueCallbacks ()
es	function unmuteValueCallbacks ()
cmd	YDataLogger target unmuteValueCallbacks

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

YAPI_SUCCESS when the call succeeds.

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

datalogger→wait_async()**YDataLogger**

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

```
js function wait_async( callback, context)
```

```
es 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.

21.5. Recorded data sequence

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

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

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

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

js	<code><script type='text/javascript' src='yocto_api.js'></script></code>
cpp	<code>#include "yocto_api.h"</code>
m	<code>#import "yocto_api.h"</code>
pas	<code>uses yocto_api;</code>
vb	<code>yocto_api.vb</code>
cs	<code>yocto_api.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YModule;</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YModule;</code>
py	<code>from yocto_api import *</code>
php	<code>require_once('yocto_api.php');</code>
es	in HTML: <code><script src='../lib/yocto_api.js'></script></code> in node.js: <code>require('yoctolib-es2017/yocto_api.js');</code>

YDataSet methods

dataset→get_endTimeUTC()

Returns the end time of the dataset, relative to the Jan 1, 1970.

dataset→get_functionId()

Returns the hardware identifier of the function that performed the measure, without reference to the module.

dataset→get_hardwareId()

Returns the unique hardware identifier of the function who performed the measures, in the form `SERIAL.FUNCTIONID`.

dataset→get_measures()

Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

dataset→get_measuresAt(measure)

Returns the detailed set of measures for the time interval corresponding to a given condensed measures previously returned by `get_preview()`.

dataset→get_preview()

Returns a condensed version of the measures that can retrieved in this YDataSet, as a list of YMeasure objects.

dataset→get_progress()

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

dataset→get_startTimeUTC()

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

dataset→get_summary()

Returns an YMeasure object which summarizes the whole DataSet.

dataset→get_unit()

Returns the measuring unit for the measured value.

dataset→loadMore()

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

dataset→loadMore_async(callback, context)

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

dataset→**get_endTimeUTC()****YDataSet****dataset**→**endTimeUTC()**

Returns the end time of the dataset, relative to the Jan 1, 1970.

js	function get_endTimeUTC ()
cpp	s64 get_endTimeUTC ()
m	-(s64) endTimeUTC
pas	function get_endTimeUTC (): int64
vb	function get_endTimeUTC () As Long
cs	long get_endTimeUTC ()
java	long get_endTimeUTC ()
uwp	Task<long> get_endTimeUTC ()
py	def get_endTimeUTC ()
php	function get_endTimeUTC ()
es	function get_endTimeUTC ()

When the YDataSet is created, the end time is the value passed in parameter to the `get_dataSet( )` function. After the very first call to `LoadMore( )`, the end time is updated to reflect the timestamp of the last measure actually found in the dataLogger within the specified range.

Returns :

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

dataset→get_functionId()**YDataSet****dataset→functionId()**

Returns the hardware identifier of the function that performed the measure, without reference to the module.

js	function get_functionId ()
cpp	string get_functionId ()
m	-(NSString*) functionId
pas	function get_functionId (): string
vb	function get_functionId () As String
cs	string get_functionId ()
java	String get_functionId ()
uwp	Task<string> get_functionId ()
py	def get_functionId ()
php	function get_functionId ()
es	function get_functionId ()

For example temperature1.

Returns :

a string that identifies the function (ex: temperature1)

dataset→get_hardwareId()**YDataSet****dataset→hardwareId()**

Returns the unique hardware identifier of the function who performed the measures, in the form SERIAL.FUNCTIONID.

js	function get_hardwareId ()
cpp	string get_hardwareId ()
m	-(NSString*) hardwareId
pas	function get_hardwareId (): string
vb	function get_hardwareId () As String
cs	string get_hardwareId ()
java	String get_hardwareId ()
uwp	Task<string> get_hardwareId ()
py	def get_hardwareId ()
php	function get_hardwareId ()
es	function 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()**YDataSet****dataset→measures()**

Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

js	function get_measures ()
cpp	vector<YMeasure> get_measures ()
m	-(NSMutableArray*) measures
pas	function get_measures (): TYMeasureArray
vb	function get_measures () As List
cs	List<YMeasure> get_measures ()
java	ArrayList<YMeasure> get_measures ()
uwp	Task<List<YMeasure>> get_measures ()
py	def get_measures ()
php	function get_measures ()
es	function get_measures ()

Each item includes: - the start of the measure time interval - the end of the measure time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

Before calling this method, you should call `LoadMore( )` to load data from the device. You may have to call `loadMore()` several time until all rows are loaded, but you can start looking at available data rows before the load is complete.

The oldest measures are always loaded first, and the most recent measures will be loaded last. As a result, timestamps are normally sorted in ascending order within the measure table, unless there was an unexpected adjustment of the datalogger UTC clock.

Returns :

a table of records, where each record depicts the measured value for a given time interval

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

dataset→get_measuresAt()**YDataSet****dataset→measuresAt()**

Returns the detailed set of measures for the time interval corresponding to a given condensed measures previously returned by `get_preview()`.

js	function get_measuresAt (measure)
cpp	vector<YMeasure> get_measuresAt (YMeasure measure)
m	-(NSMutableArray*) measuresAt : (YMeasure*) measure
pas	function get_measuresAt (measure : TYMeasure): TYMeasureArray
vb	function get_measuresAt () As List
cs	List<YMeasure> get_measuresAt (YMeasure measure)
java	ArrayList<YMeasure> get_measuresAt (YMeasure measure)
uwp	Task<List<YMeasure>> get_measuresAt (YMeasure measure)
py	def get_measuresAt (measure)
php	function get_measuresAt (\$measure)
es	function get_measuresAt (measure)

The result is provided as a list of YMeasure objects.

Parameters :

measure condensed measure from the list previously returned by `get_preview()`.

Returns :

a table of records, where each record depicts the measured values during a time interval

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

dataset→get_preview()**YDataSet****dataset→preview()**

Returns a condensed version of the measures that can retrieved in this YDataSet, as a list of YMeasure objects.

js	function get_preview ()
cpp	vector<YMeasure> get_preview ()
m	-(NSMutableArray*) preview
pas	function get_preview (): TYMeasureArray
vb	function get_preview () As List
cs	List<YMeasure> get_preview ()
java	ArrayList<YMeasure> get_preview ()
uwp	Task<List<YMeasure>> get_preview ()
py	def get_preview ()
php	function get_preview ()
es	function 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()**YDataSet****dataset→progress()**

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

js	function get_progress ()
cpp	int get_progress ()
m	-(int) progress
pas	function get_progress (): LongInt
vb	function get_progress () As Integer
cs	int get_progress ()
java	int get_progress ()
uwp	Task<int> get_progress ()
py	def get_progress ()
php	function get_progress ()
es	function get_progress ()

When the object is instantiated by `get_dataSet`, the progress is zero. Each time `loadMore( )` is invoked, the progress is updated, to reach the value 100 only once all measures have been loaded.

Returns :

an integer in the range 0 to 100 (percentage of completion).

dataset→**get_startTimeUTC()****YDataSet****dataset**→**startTimeUTC()**

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

js	function get_startTimeUTC ()
cpp	s64 get_startTimeUTC ()
m	-(s64) startTimeUTC
pas	function get_startTimeUTC (): int64
vb	function get_startTimeUTC () As Long
cs	long get_startTimeUTC ()
java	long get_startTimeUTC ()
uwp	Task<long> get_startTimeUTC ()
py	def get_startTimeUTC ()
php	function get_startTimeUTC ()
es	function get_startTimeUTC ()

When the YDataSet is created, the start time is the value passed in parameter to the `get_dataSet( )` function. After the very first call to `loadMore( )`, the start time is updated to reflect the timestamp of the first measure actually found in the dataLogger within the specified range.

Returns :

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

dataset→get_summary()**YDataSet****dataset→summary()**

Returns an YMeasure object which summarizes the whole DataSet.

js	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 ()
uwp	Task<YMeasure> get_summary ()
py	def get_summary ()
php	function get_summary ()
es	function get_summary ()

It includes the following information: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This summary is available as soon as `LoadMore( )` has been called for the first time.

Returns :

an YMeasure object

dataset→get_unit()**YDataSet****dataset→unit()**

Returns the measuring unit for the measured value.

js	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 ()
uwp	Task<string> get_unit ()
py	def get_unit ()
php	function get_unit ()
es	function get_unit ()

Returns :

a string that represents a physical unit.

On failure, throws an exception or returns Y_UNIT_INVALID.

dataset→loadMore()**YDataSet**

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

js	function loadMore ()
cpp	int loadMore ()
m	-(int) loadMore
pas	function loadMore (): LongInt
vb	function loadMore () As Integer
cs	int loadMore ()
java	int loadMore ()
uwp	Task<int> loadMore ()
py	def loadMore ()
php	function loadMore ()
es	function loadMore ()

Returns :

an integer in the range 0 to 100 (percentage of completion), or a negative error code in case of failure.

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

dataset→loadMore_async()**YDataSet**

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

```
js function loadMore_async( callback, context)
```

Parameters :

callback callback function that is invoked when the w The callback function receives three arguments: - the user-specific context object - the YDataSet object whose loadMore_async was invoked - the load result: either the progress indicator (0...100), or a negative error code in case of failure.

context user-specific object that is passed as-is to the callback function

Returns :

nothing.

21.6. 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	<code><script type='text/javascript' src='yocto_api.js'></script></code>
cpp	<code>#include "yocto_api.h"</code>
m	<code>#import "yocto_api.h"</code>
pas	<code>uses yocto_api;</code>
vb	<code>yocto_api.vb</code>
cs	<code>yocto_api.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YModule;</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YModule;</code>
py	<code>from yocto_api import *</code>
php	<code>require_once('yocto_api.php');</code>
es	in HTML: <code><script src='../lib/yocto_api.js'></script></code> in node.js: <code>require('yoctolib-es2017/yocto_api.js');</code>

YMeasure methods

measure→get_averageValue()

Returns the average value observed during the time interval covered by this measure.

measure→get_endTimeUTC()

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

measure→get_maxValue()

Returns the largest value observed during the time interval covered by this measure.

measure→get_minValue()

Returns the smallest value observed during the time interval covered by this measure.

measure→get_startTimeUTC()

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

measure→get_averageValue()**YMeasure****measure→averageValue()**

Returns the average value observed during the time interval covered by this measure.

js	function get_averageValue ()
cpp	double get_averageValue ()
m	-(double) averageValue
pas	function get_averageValue (): double
vb	function get_averageValue () As Double
cs	double get_averageValue ()
java	double get_averageValue ()
uwp	double get_averageValue ()
py	def get_averageValue ()
php	function get_averageValue ()
es	function get_averageValue ()

Returns :

a floating-point number corresponding to the average value observed.

measure→**get_endTimeUTC()****YMeasure****measure**→**endTimeUTC()**

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

js	function get_endTimeUTC ()
cpp	double get_endTimeUTC ()
m	-(double) endTimeUTC
pas	function get_endTimeUTC (): double
vb	function get_endTimeUTC () As Double
cs	double get_endTimeUTC ()
java	double get_endTimeUTC ()
uwp	double get_endTimeUTC ()
py	def get_endTimeUTC ()
php	function get_endTimeUTC ()
es	function 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 ()
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 ()
uwp	double get_maxValue ()
py	def get_maxValue ()
php	function get_maxValue ()
es	function get_maxValue ()

Returns :

a floating-point number corresponding to the largest value observed.

measure→**get_minValue()****YMeasure****measure**→**minValue()**

Returns the smallest value observed during the time interval covered by this measure.

js	function get_minValue ()
cpp	double get_minValue ()
m	-(double) minValue
pas	function get_minValue (): double
vb	function get_minValue () As Double
cs	double get_minValue ()
java	double get_minValue ()
uwp	double get_minValue ()
py	def get_minValue ()
php	function get_minValue ()
es	function get_minValue ()

Returns :

a floating-point number corresponding to the smallest value observed.

measure→get_startTimeUTC()**YMeasure****measure→startTimeUTC()**

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

js	function get_startTimeUTC ()
cpp	double get_startTimeUTC ()
m	-(double) startTimeUTC
pas	function get_startTimeUTC (): double
vb	function get_startTimeUTC () As Double
cs	double get_startTimeUTC ()
java	double get_startTimeUTC ()
uwp	double get_startTimeUTC ()
py	def get_startTimeUTC ()
php	function get_startTimeUTC ()
es	function get_startTimeUTC ()

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.

22. Troubleshooting

22.1. Where to start?

If it is the first time that you use a Yoctopuce module and you do not really know where to start, have a look at the Yoctopuce blog. There is a section dedicated to beginners ¹.

22.2. Programming examples don't seem to work

Most of Yoctopuce API programming examples are command line programs and require some parameters to work properly. You have to start them from your operating system command prompt, or configure your IDE to run them with the proper parameters. ².

22.3. Linux and USB

To work correctly under Linux, the the library needs to have write access to all the Yoctopuce USB peripherals. However, by default under Linux, USB privileges of the non-root users are limited to read access. To avoid having to run the *VirtualHub* as root, you need to create a new *udev* rule to authorize one or several users to have write access to the Yoctopuce peripherals.

To add a new *udev* rule to your installation, you must add a file with a name following the "`##-arbitraryName.rules`" format, in the `/etc/udev/rules.d` directory. When the system is starting, *udev* reads all the files with a `".rules"` extension in this directory, respecting the alphabetical order (for example, the `"51-custom.rules"` file is interpreted AFTER the `"50-udev-default.rules"` file).

The `"50-udev-default"` file contains the system default *udev* rules. To modify the default behavior, you therefore need to create a file with a name that starts with a number larger than 50, that will override the system default rules. Note that to add a rule, you need a root access on the system.

In the `udev_conf` directory of the *VirtualHub* for Linux³ archive, there are two rule examples which you can use as a basis.

¹ see: http://www.yoctopuce.com/EN/blog_by_categories/for-the-beginners

² see: <http://www.yoctopuce.com/EN/article/about-programming-examples>

³ <http://www.yoctopuce.com/FR/virtualhub.php>

Example 1: 51-yoctopuce.rules

This rule provides all the users with read and write access to the Yoctopuce USB peripherals. Access rights for all other peripherals are not modified. If this scenario suits you, you only need to copy the "51-yoctopuce_all.rules" file into the "/etc/udev/rules.d" directory and to restart your system.

```
# udev rules to allow write access to all users
# for Yoctopuce USB devices
SUBSYSTEM=="usb", ATTR{idVendor}=="24e0", MODE="0666"
```

Example 2: 51-yoctopuce_group.rules

This rule authorizes the "yoctogroup" group to have read and write access to Yoctopuce USB peripherals. Access rights for all other peripherals are not modified. If this scenario suits you, you only need to copy the "51-yoctopuce_group.rules" file into the "/etc/udev/rules.d" directory and restart your system.

```
# udev rules to allow write access to all users of "yoctogroup"
# for Yoctopuce USB devices
SUBSYSTEM=="usb", ATTR{idVendor}=="24e0", MODE="0664", GROUP="yoctogroup"
```

22.4. ARM Platforms: HF and EL

There are two main flavors of executable on ARM: HF (Hard Float) binaries, and EL (EABI Little Endian) binaries. These two families are not compatible at all. The compatibility of a given ARM platform with 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.

22.5. Powered module but invisible for the OS

If your Yocto-CO2 is connected by USB, if its blue led is on, but if the operating system cannot see the module, check that you are using a true USB cable with data wires, and not a charging cable. Charging cables have only power wires.

22.6. Another process named xxx is already using yAPI

If when initializing the Yoctopuce API, you obtain the *"Another process named xxx is already using yAPI"* error message, it means that another application is already using Yoctopuce USB modules. On a single machine only one process can access Yoctopuce modules by USB at a time. You can easily work around this limitation by using a VirtualHub and the network mode ⁴.

22.7. Disconnections, erratic behavior

If your Yocto-CO2 behaves erratically and/or disconnects itself from the USB bus without apparent reason, check that it is correctly powered. Avoid cables with a length above 2 meters. If needed, insert a powered USB hub ^{5 6}.

⁴ see: <http://www.yoctopuce.com/EN/article/error-message-another-process-is-already-using-yapi>

⁵ see: <http://www.yoctopuce.com/EN/article/usb-cables-size-matters>

⁶ see: <http://www.yoctopuce.com/EN/article/how-many-usb-devices-can-you-connect>

22.8. Damaged device

Yoctopuce strives to reduce the production of electronic waste. If you believe that your Yocto-CO2 is not working anymore, start by contacting Yoctopuce support by e-mail to diagnose the failure. Even if you know that the device was damaged by mistake, Yoctopuce engineers might be able to repair it, and thus avoid creating electronic waste.



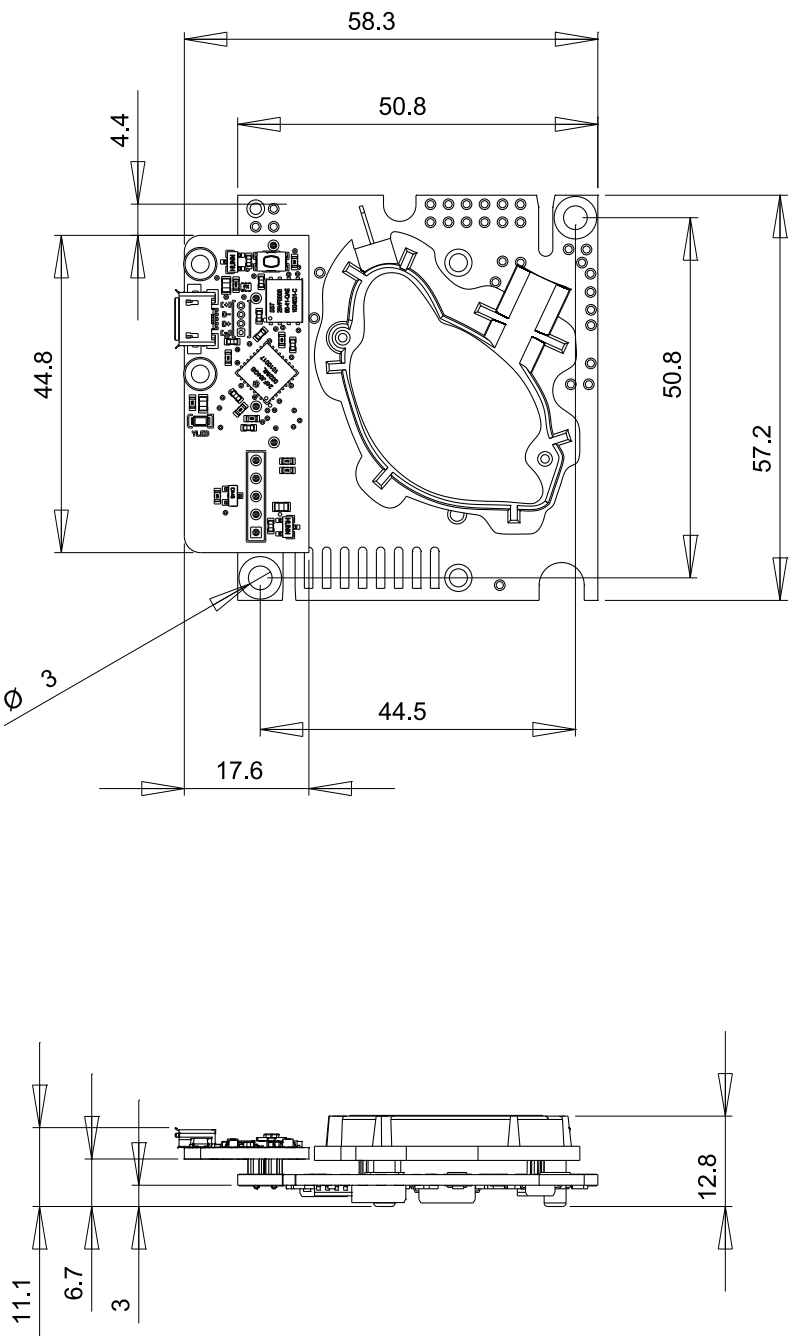
Waste Electrical and Electronic Equipment (WEEE) If you really want to get rid of your Yocto-CO2, do not throw it away in a trash bin but bring it to your local WEEE recycling point. In this way, it will be disposed properly by a specialized WEEE recycling center.



23. Characteristics

You can find below a summary of the main technical characteristics of your Yocto-CO2 module.

USB connector	micro-B
Thickness	12 mm
Width	57 mm
Length	58 mm
Weight	18.6 g
Sensor	SenseAir K-30
Refresh rate	1 Hz
Measuring range	0-10000 ppm (vol)
Accuracy	30 ppm 5%
Sensitivity	20 ppm 1%
IEC protection class	class III
Normal operating temperature	5...40 °C
Extended operating temperature	0...50 °C
Supported Operating Systems	Windows (PC + IoT), Linux (Intel + ARM), macOS, Android
Drivers	no driver needed
API / SDK / Libraries	C++, Objective-C, C#, VB.NET, UWP, Delphi, Python, Java, Android
API / SDK / Libraries (TCP only)	Javascript, Node.js, PHP
RoHS	Yes
USB Vendor ID	0x24E0
USB Device ID	0x0027
Suggested enclosure	YoctoBox-CO2-Black
Cables and enclosures	available separately



All dimensions are in mm
Toutes les dimensions sont en mm

Yocto-CO2

A4
Scale
1:1
Echelle