



Yocto-4-20mA-Rx, User's guide

Table of contents

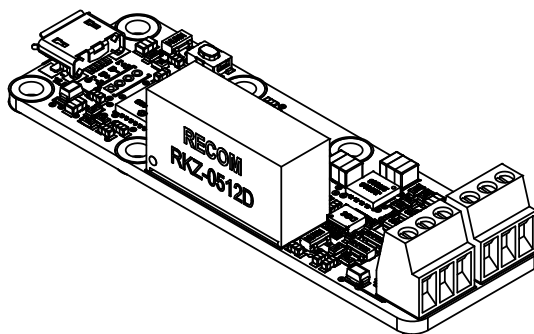
1. Introduction	1
1.1. Prerequisites	1
1.2. Optional accessories	3
2. Presentation	5
2.1. Common elements	5
2.2. Specific elements	6
3. First steps	11
3.1. Localization	11
3.2. Test of the module	11
3.3. Configuration	12
4. Assembly and connections	15
4.1. Fixing	15
4.2. USB power distribution	16
5. Programming, general concepts	17
5.1. Programming paradigm	17
5.2. The Yocto-4-20mA-Rx module	18
5.3. Module control interface	19
5.4. GenericSensor function interface	20
5.5. DataLogger function interface	21
5.6. What interface: Native, DLL or Service ?	22
5.7. Programming, where to start?	24
6. Using the Yocto-4-20mA-Rx in command line	27
6.1. Installing	27
6.2. Use: general description	27
6.3. Control of the GenericSensor function	28
6.4. Control of the module part	28
6.5. Limitations	29
7. Using Yocto-4-20mA-Rx with Javascript	31
7.1. Getting ready	31

7.2. Control of the GenericSensor function	31
7.3. Control of the module part	33
7.4. Error handling	36
8. Using Yocto-4-20mA-Rx with PHP	39
8.1. Getting ready	39
8.2. Control of the GenericSensor 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-4-20mA-Rx with C++	49
9.1. Control of the GenericSensor 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-4-20mA-Rx with Objective-C	57
10.1. Control of the GenericSensor function	57
10.2. Control of the module part	59
10.3. Error handling	61
11. Using Yocto-4-20mA-Rx 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 GenericSensor function	64
11.4. Control of the module part	66
11.5. Error handling	68
12. Using Yocto-4-20mA-Rx with C#	69
12.1. Installation	69
12.2. Using the Yoctopuce API in a Visual C# project	69
12.3. Control of the GenericSensor function	70
12.4. Control of the module part	72
12.5. Error handling	74
13. Using Yocto-4-20mA-Rx with Delphi	77
13.1. Preparation	77
13.2. Control of the GenericSensor function	77
13.3. Control of the module part	79
13.4. Error handling	81
14. Using the Yocto-4-20mA-Rx with Python	83
14.1. Source files	83
14.2. Dynamic library	83
14.3. Control of the GenericSensor function	83
14.4. Control of the module part	85
14.5. Error handling	87
15. Using the Yocto-4-20mA-Rx with Java	89
15.1. Getting ready	89
15.2. Control of the GenericSensor function	89
15.3. Control of the module part	91
15.4. Error handling	93

16. Using the Yocto-4-20mA-Rx with Android	95
16.1. Native access and VirtualHub	95
16.2. Getting ready	95
16.3. Compatibility	95
16.4. Activating the USB port under Android	96
16.5. Control of the GenericSensor function	98
16.6. Control of the module part	100
16.7. Error handling	105
17. Advanced programming	107
17.1. Event programming	107
17.2. The data logger	110
17.3. Sensor calibration	112
18. Using with unsupported languages	117
18.1. Command line	117
18.2. VirtualHub and HTTP GET	117
18.3. Using dynamic libraries	119
18.4. Porting the high level library	122
19. High-level API Reference	123
19.1. General functions	124
19.2. Module control interface	148
19.3. GenericSensor function interface	193
19.4. Recorded data sequence	243
19.5. Measured value	256
19.6. Unformatted data sequence	262
20. Troubleshooting	277
20.1. Linux and USB	277
20.2. ARM Platforms: HF and EL	278
21. Characteristics	279
Blueprint	281
Index	283

1. Introduction

The Yocto-4-20mA-Rx is a 60x20mm module which can interface two sensors following the 4-20mA standard. The Yocto-4-20mA-Rx is also able to power the sensors with a 23V voltage, up to 80mA. The Yocto-4-20mA-Rx was conceived as an electrically insulated interface: there is a galvanic insulation between the measure and the USB parts of the module. This enables you to work with sensors which are not at the same potential as the computer driving the Yocto-4-20mA-Rx.



The Yocto-4-20mA-Rx module

Yoctopuce thanks you for buying this Yocto-4-20mA-Rx 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-4-20mA-Rx is easy to install anywhere and easy to drive from a maximum of programming languages. If you are nevertheless disappointed with this module, do not hesitate to contact Yoctopuce support¹.

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

1.1. Prerequisites

In order to use your Yocto-4-20mA-Rx 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.

¹ support@yoctopuce.com

Yoctopuce provides software libraries to drive its modules for the following operating systems: Windows, Mac OS X, Linux, and Android. Yoctopuce modules do not require installing any specific system driver, as they leverage the standard HID driver² provided with every operating system.

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

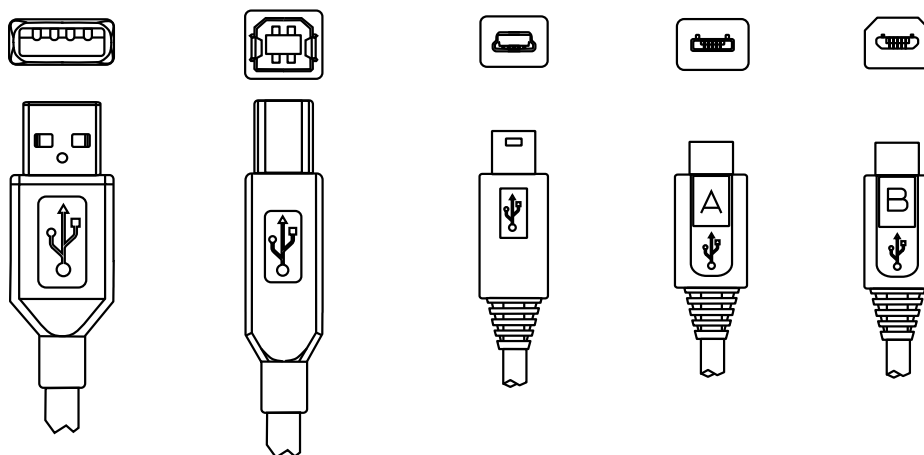
Mac OS X versions currently supported are: 10.6 (Snow Leopard), Mac OS X 10.7 (Lion), and 10.8 (Mountain Lion). Yoctopuce is frequently testing its modules on Mac OS X 10.6 and 10.7.

Linux kernels currently supported are the 2.6 branch and the 3.0 branch. Other versions of the Linux kernel, and even other UNIX variants, are very likely to work as well, as Linux support is implemented through the standard **libusb** API. Yoctopuce is frequently testing its modules on Linux kernel 2.6.

Android versions currently supported are: Android 3.1 and later. Moreover, it is necessary for the tablet or phone to support the *Host* USB mode. Yoctopuce is frequently testing its modules on Android 4.x on a Nexus 7 and a Samsung Galaxy S3 with the Java for Android library.

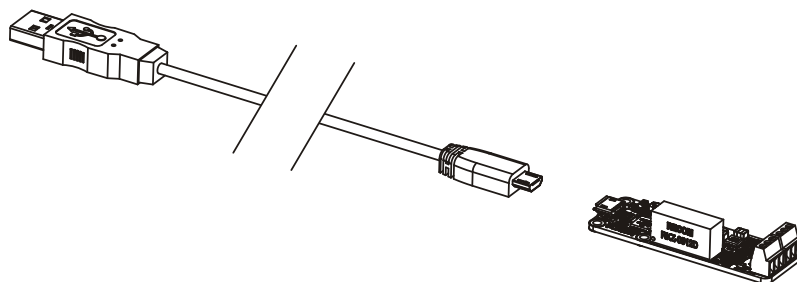
A USB cable, type A-micro B

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



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

To connect your Yocto-4-20mA-Rx module to a computer, you need a USB cable of type A-micro B. The price of this cable may vary a lot depending on the source, look for it under the name *USB A to micro B 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-4-20mA-Rx module with a USB cable of type A - micro B.

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

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

If you insert a USB hub between the computer and the Yocto-4-20mA-Rx module, make sure to take into account the USB current limits. If you do not, be prepared to face unstable behaviors and unpredictable failures. You can find more details on this topic in the chapter about assembly and connections.

1.2. Optional accessories

The accessories below are not necessary to use the Yocto-4-20mA-Rx module but might be useful depending on your project. These are mostly common products that you can buy from your favourite hacking store. To save you the tedious job of looking for them, most of them are also available on the Yoctopuce shop.

Screws and spacers

In order to mount the Yocto-4-20mA-Rx module, you can put small screws in the 2.5mm assembly holes, with a screw head no larger than 4.5mm. The best way is to use threaded spacers, which you can then mount wherever you want. You can find more details on this topic in the chapter about assembly and connections.

Micro-USB hub

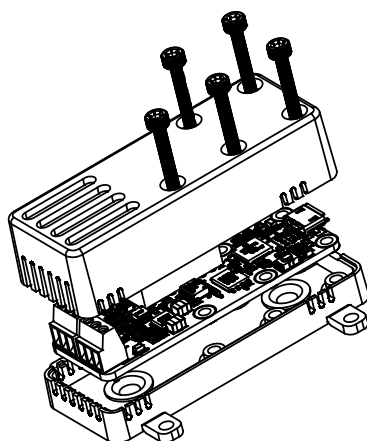
If you intend to put several Yoctopuce modules in a very small space, you can connect them directly to a micro-USB hub. Yoctopuce builds a USB hub particularly small for this purpose (down to 20mmx36mm), on which you can directly solder a USB cable instead of using a USB plug. For more details, see the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

YoctoHub-Ethernet and YoctoHub-Wireless

You can add network connectivity to your Yocto-4-20mA-Rx, thanks to the YoctoHub-Ethernet and the YoctoHub-Wireless. The YoctoHub-Ethernet provides Ethernet connectivity and the YoctoHub-Wireless provides WiFi connectivity. Both can drive up to three devices and behave exactly like a regular computer running a *VirtualHub*.

Enclosures

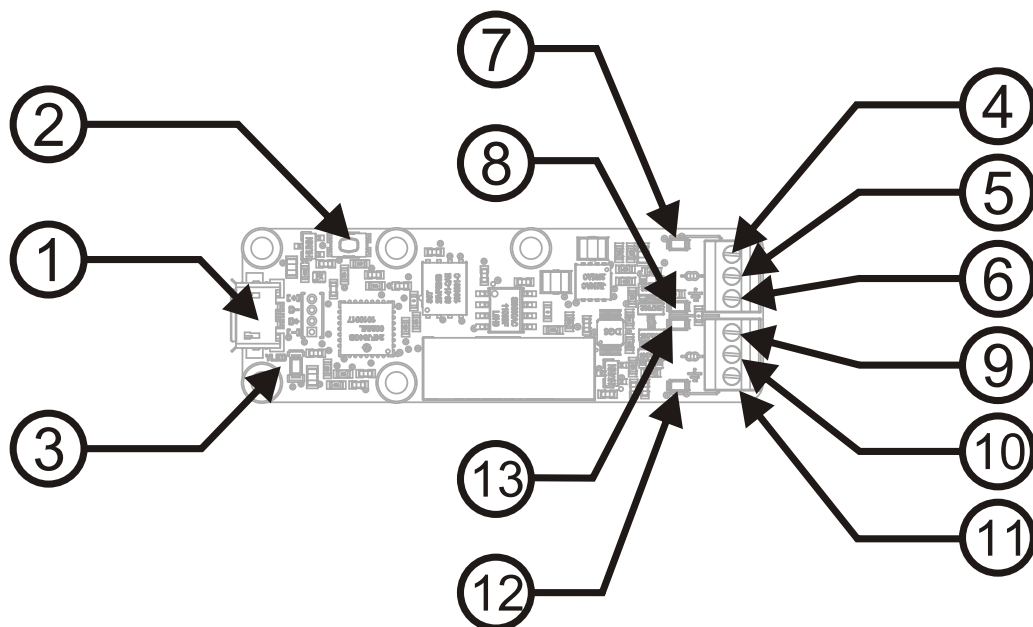
Your Yocto-4-20mA-Rx has been designed to be installed as is in your project. Nevertheless, Yoctopuce sells enclosures specifically designed for Yoctopuce devices. These enclosures have removable mounting brackets and magnets allowing them to stick on ferromagnetic surfaces. More details are available on the Yoctopuce web site ⁴. The suggested enclosure model for your Yocto-4-20mA-Rx is the YoctoBox-Long-Thick-Black-Vents.



You can install your Yocto-4-20mA-Rx in an optional enclosure

⁴ <http://www.yoctopuce.com/EN/products/category/enclosures>

2. Presentation



- | | |
|--------------------------|---------------------------------|
| 1: Micro-B USB socket | 7: Sensor 1 led |
| 2: Yocto-button | 8: Sensor 1 input overload led |
| 3: Yocto-led | 9: Sensor 2 power supply |
| 4: Sensor 1 power supply | 10: Sensor 2 current loop |
| 5: Sensor 1 current loop | 11: Sensor 2 ground |
| 6: Sensor 1 ground | 12: Sensor 2 led |
| | 13: Sensor 2 input overload led |

2.1. Common elements

All Yocto-modules share a number of common functionalities.

USB connector

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

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

Yocto-button

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

Yocto-led

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

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

The Yocto-led has a third functionality, which is less pleasant: when the internal software which controls the module encounters a fatal error, the Yocto-led starts emitting an SOS in 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-4-20mA-Rx modules, this number starts with RX420MA1. 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

Inputs

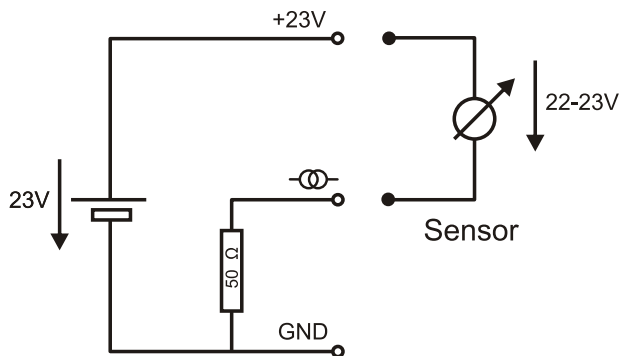
There are two inputs in the Yocto-4-20mA-Rx to connect sensors following the 4-20mA standard. As implied by their name, these sensors have the specificity to transmit the measured values using a

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

² support@yoctopuce.com

current loop, regulating a current varying linearly between 4 and 20 mA depending on the measure. The Yocto-4-20mA-Rx is essentially a precision mini-ammeter able to measure this current.

The current loop used to transmit the measure starts from the pole marked 23V on the Yocto-4-20mA-Rx, goes through the sensor which controls the current, and returns to the Yocto-4-20mA-Rx on the pole marked with a loop sign. Here is a simplified diagram of the electrical interface:



Yocto-4-20mA-RX

Electrical diagram equivalent to the input of the Yocto-4-20mA-Rx

The Yocto-4-20mA-Rx performs its current measure with the help of a 50 Ohm resistance, which limits the voltage drop of the measure to 1V at the maximum with regards to the module ground. About 22V remain available for the current loop. Most sensors require a minimum 12V to work, which leaves a 10V or so margin to overcome the electric cable resistance between the Yocto-4-20mA-Rx and the sensor. This allows a distance of more than 2500 meters using AWG 24 cable.

Automatic conversion

The Yocto-4-20mA-Rx is able to automatically convert the measured current into the physical quantity measured by the sensor. The conversion is a simple linear conversion based on the mapping of the [4mA...20mA] range and the range of extreme values that the sensor is able to measure. You can configure this mapping in the Yocto-4-20mA-Rx, with the help of the *VirtualHub* for example. You can extend the measure range up to 0mA, but it is useful to keep a non-null minimal value to detect sensor connection errors.

The green led

Each Yocto-4-20mA-Rx input is equipped with a green led. Its intensity provides an indication of the value read on the current loop. This led is off when the sensor is not properly connected.

Sensor power supply

To power themselves, many 4..20mA sensors simply take the necessary energy on the current loop, using the always available 4mA. Some sensors require a little more power. The Yocto-4-20mA-Rx can provide it with the help of the 23V power supply regulated between the 23V pole and the pole marked with the ground sign. Beware, the current available for both sensors together is a total of 80mA, including current used in the loop. If you go above this value, the behavior of the 4..20mA sensor will probably be modified and you even risk to damage the 23V power supply of the Yocto-4-20mA-Rx. If your sensors require more power to work, you must use an independent power supply (see diagrams below).

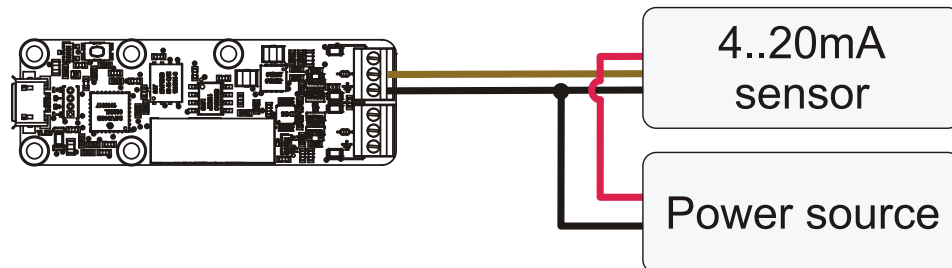
When the 23V power supply of the Yocto-4-20mA-Rx is in use, the Yocto-4-20mA-Rx starts to heat. This is a normal behavior. Make sure that this heat can be evacuated.



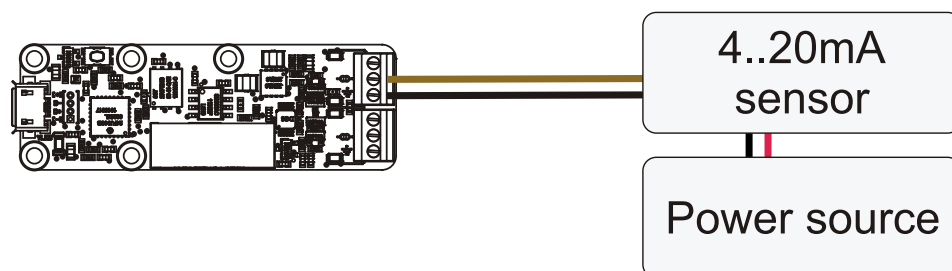
Wiring for a 2 wire sensor, powered by the current loop.



Wiring for a 3 wire sensor, powered by the Yocto-4-20mA-Rx (max. 80mA in all).



Wiring for a 3 wire sensor, with an independent power supply with common ground.



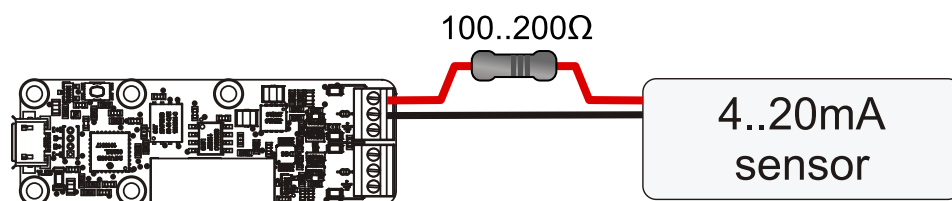
Wiring of a 4 wire sensor, with an independent power supply.

We recommend that you turn off the Yocto-4-20mA-Rx before you connect a sensor. If this is not possible and you use a 3 wire sensor, make sure to always connect the ground of the sensor first. Otherwise, you risk to provoke an overload on the measuring loop, which makes the Yocto-4-20mA-Rx go into protection mode.

The overload (red) led

The Yocto-4-20mA-Rx was not designed to measure more than 20mA. If a higher intensity is detected in the current loop, because of a short for instance, the corresponding input goes into protection mode and the corresponding red led is switched on. If the Yocto-4-20mA-Rx reports values above 20mA, you must consider them to be inaccurate.

The Yocto-4-20mA-Rx internal resistance is only 50Ω, this allows the use of very long wire for the 4-20mA sensor. However, in some cases, the low resistance combined with sensors using big capacitors may cause protection problems at power up: the capacitors charging process will cause a current rush greater than 20mA and trigger the protection mode. This can be avoided by adding a 100-200Ω resistor on the sensor power supply wire.



A small resistor can avoid spurious protection triggering at power-up.

Insulation

The measuring part of the Yocto-4-20mA-Rx is electrically insulated from the USB part. This means that you can use sensors with external power supplies which do not share the same electrical

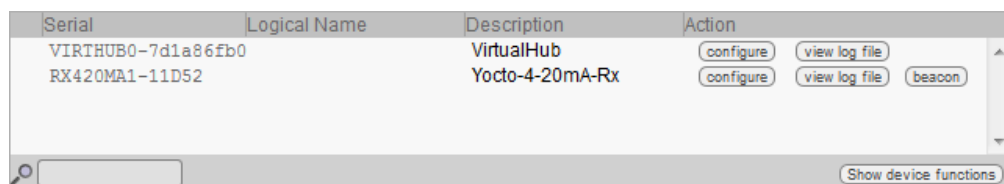
potential³ as your driving computer without risking to damage your equipment. However, the two inputs are not insulated from one another, they share a common ground.

³ This kind of situation happens in particular when separate parts of an installation, connected to the mains, are powered by distinct phases.

3. First steps

When reading this chapter, your Yocto-4-20mA-Rx should be connected to your computer, which should have recognized it. It is time to make it work.

Go to the Yoctopuce web site and download the *Virtual Hub* software¹. It is available for Windows, Linux, and Mac OS X. Normally, the Virtual Hub software serves as an abstraction layer for languages which cannot access the hardware layers of your computer. However, it also offers a succinct interface to configure your modules and to test their basic functions. You access this interface with a simple web browser². Start the *Virtual Hub* software in a command line, open your preferred web browser and enter the URL `http://127.0.0.1:4444`. The list of the Yoctopuce modules connected to your computer is displayed.



Serial	Logical Name	Description	Action
VIRTHUB0-7d1a86fb0		VirtualHub	<button>configure</button> <button>view log file</button>
RX420MA1-11D52		Yocto-4-20mA-Rx	<button>configure</button> <button>view log file</button> <button>beacon</button>

Below the table is a search bar with a magnifying glass icon and a "Show device functions" button.

Module list as displayed in your web browser.

3.1. Localization

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

3.2. Test of the module

The first item to check is that your module is working well: click on the serial number corresponding to your module. This displays a window summarizing the properties of your Yocto-4-20mA-Rx.

¹ www.yoctopuce.com/EN/virtualhub.php

² The interface was tested on FireFox 3+, IE 6+, Safari, and Chrome. It does not work with Opera.

Kernel

Serial #
Product name: Yocto-4-20mA-Rx
Logical name
Product release: 1
Firmware: 12696
Consumption: 231 mA
Beacon: Inactive turn on
Luminosity: 50%

Sensors

	Min	Current	Max
Sensor 1:	237.2 deg	237.6 deg (14.558 mA)	2372.7 deg
Sensor 2:	20 mA	--- (0 mA)	4 mA

Misc

Open API browser (pop-up)
Get user manual from yoctopuce.com

Close

Properties of the Yocto-4-20mA-Rx module.

This window allows you, among other things, to play with your module to check that it is working properly. Values measured by the Yocto-4-20mA-Rx are indeed displayed in real time.

3.3. Configuration

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

Edit parameters for device RX420MA1-11D52, and click on the **Save** button.

Serial #
Product name: Yocto-4-20mA-Rx
Firmware: 12696 upgrade
Logical name
Luminosity: slider (signal leds only)

Device functions

Each function of the device has a physical name and a logical name. You can change the logical name using the **rename** button. You can map the signal read by the device to any physical measure in the range -25000.000 ... +25000.000

RX420MA1-11D52.genericSensor1 / rename

Mapping type: 4...20mA
Mapped value for 4mA: 0
Mapped value for 20mA: 360
Mapped value unit: deg
Display resolution: 0.1

RX420MA1-11D52.genericSensor2 / rename

Mapping type: 4...20mA
Mapped value for 4mA: 4
Mapped value for 20mA: 20
Mapped value unit: mA
Display resolution: 0.001

Save Cancel

Yocto-4-20mA-Rx module configuration.

Firmware

The module firmware can easily be updated with the help of the interface. To do so, you must beforehand have the adequate firmware on your local disk. Firmware destined for Yoctopuce modules are available as .byn files and can be downloaded from the Yoctopuce web site.

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

However, it is always possible to reprogram the module correctly by using the *Virtual Hub* software³ in command line⁴.

Logical name of the module

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

Luminosity

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

Logical names of functions

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

The functions provided by the Yocto-4-20mA-Rx module are the two "genericSensor1" and "genericSensor2" functions, corresponding to the two channels. Simply click on the corresponding "rename" buttons to assign them new logical names.

mA to physical quantity conversion

You can define on which basis you want to perform the conversion between the current measured in mA and the physical quantity measured by the sensor connected to the Yocto-4-20mA-Rx. You can also define a conversion based on a range other than 4..20mA. Some exotic sensors work with other ranges, such as 1..20mA, for example.

³ www.yoctopuce.com/EN/virtualhub.php

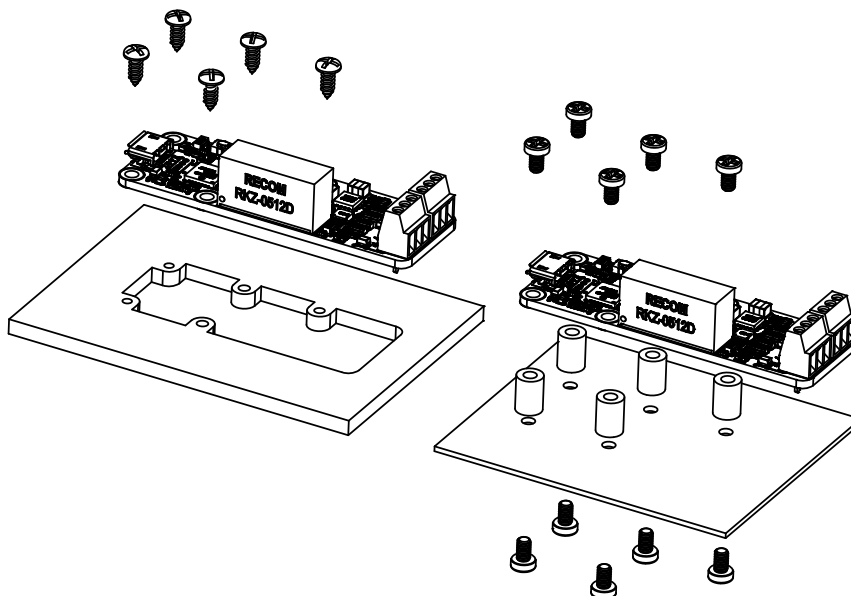
⁴ More information available in the virtual hub documentation

4. Assembly and connections

This chapter provides important information regarding the use of the Yocto-4-20mA-Rx 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-4-20mA-Rx module contains 2.5mm assembly holes. You can use these holes for screws. The screw head diameter must not be larger than 4.5mm or they will damage the module circuits. Make sure that the lower surface of the module is not in contact with the support. We recommend using spacers, but other methods are possible. Nothing prevents you from fixing the module with a glue gun; it will not be good-looking, but it will hold.

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

While working, the Yocto-4-20mA-Rx can heat up, in particular when using its 23V power supply. This is a normal behavior. When fixing your module, make sure that the heat cannot accumulate. Avoid hermetically closed enclosures.

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

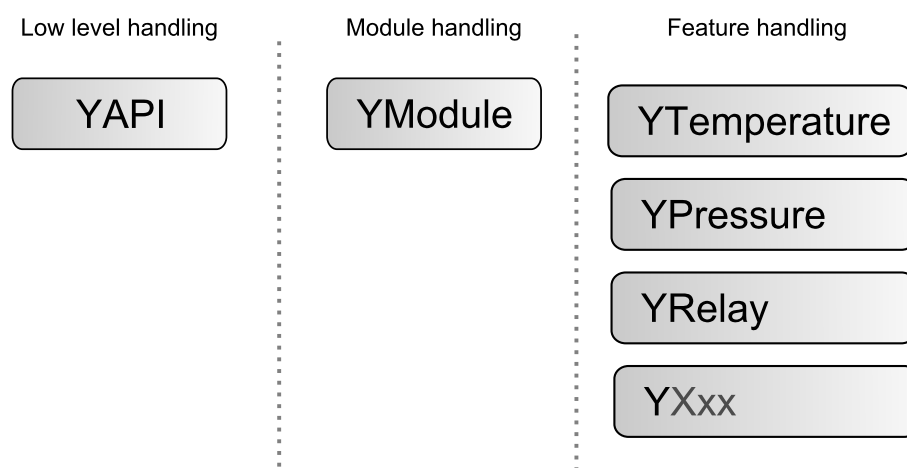
5. Programming, general concepts

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-4-20mA-Rx 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.

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-4-20mA-Rx 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-4-20mA-Rx module

The Yocto-4-20mA-Rx module provides two instances of the genericSensor function, each based on the measure of one of the two current loops used to connect external 4-20mA sensors.

module : Module

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

genericSensor1 : GenericSensor
genericSensor2 : GenericSensor

attribute	type	modifiable ?
logicalName	String	modifiable
advertisedValue	String	read-only
unit	String	modifiable
currentValue	Fixed-point number	read-only
lowestValue	Fixed-point number	modifiable
highestValue	Fixed-point number	modifiable
currentRawValue	Fixed-point number	read-only
logFrequency	Frequency	modifiable
reportFrequency	Frequency	modifiable
calibrationParam	16 bit word array	modifiable
resolution	Floating-point number	modifiable
signalValue	Fixed-point number	read-only
signalUnit	String	read-only
signalRange	Value range	modifiable
valueRange	Value range	modifiable

dataLogger : DataLogger

attribute	type	modifiable ?
logicalName	String	modifiable
advertisedValue	String	read-only
currentRunIndex	Integer	read-only
timeUTC	UTC time	modifiable
recording	On/Off	modifiable
autoStart	On/Off	modifiable
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-4-20mA-Rx module, this serial number always starts with RX420MA1. 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 55 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.

usbBandwidth

Number of USB interfaces used by the device. If this parameter is set to **DOUBLE**, the device can send twice as much data, but this may saturate the USB hub. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

5.4. GenericSensor function interface

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

logicalName

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

unit

Short character string representing the measuring unit for the measured value.

currentValue

Current value of the physical value measured by the sensor, as a floating point number.

lowestValue

Minimal value of the physical value measured by the sensor, as a floating point number.

highestValue

Maximal value of the physical value measured by the sensor, as a floating point number.

logFrequency

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

reportFrequency

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

calibrationParam

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

resolution

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

signalValue

Current value of the electrical signal generated by the sensor, as a floating point number.

signalUnit

Short character string representing the measuring unit of the electrical signal used by the sensor.

signalRange

Electric signal range used by the sensor.

valueRange

Physical value range measured by the sensor, used to convert the signal.

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.

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.

clearHistory

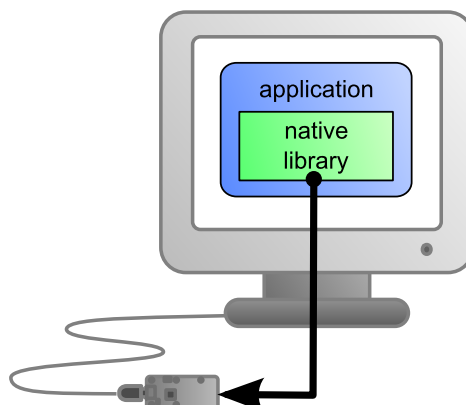
Attribute that can be set to true to clear recorded data.

5.6. What interface: Native, DLL or Service ?

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

Native control

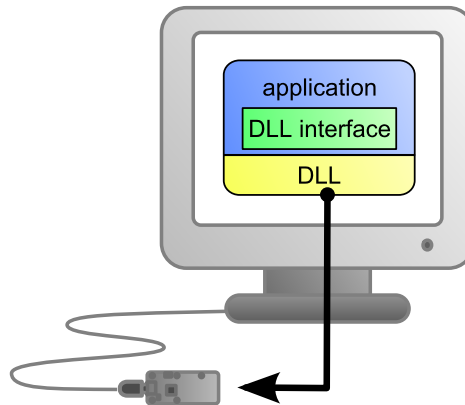
In this case, the software driving your project is compiled directly with a library which provides control of the modules. Objectively, it is the simplest and most elegant solution for the end user. The end user then only needs to plug the USB cable and run your software for everything to work. Unfortunately, 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

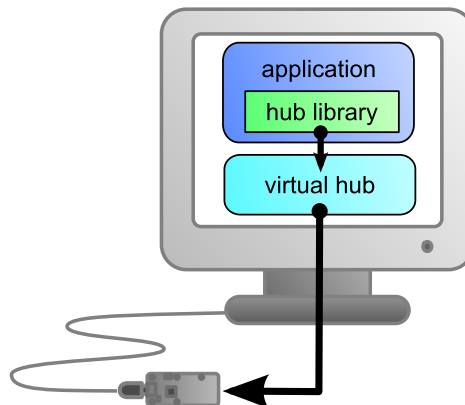
Here, the main part of the code controlling the modules is located in a DLL. The software is compiled with a small library which provides control of the DLL. It is the fastest method to code module support in a given language. Indeed, the "useful" part of the control code is located in the DLL which 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

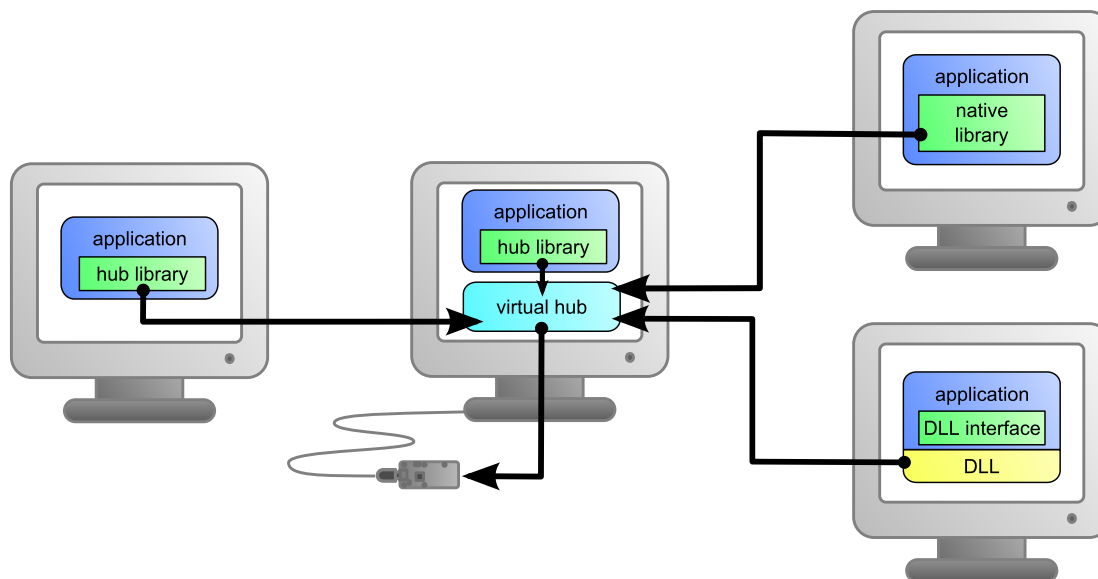
Some languages do simply not allow you to easily gain access to the hardware layers of the machine. It is the case for Javascript, for instance. To deal with this case, Yoctopuce provides a solution in the form of a small piece of software called *Virtual Hub*¹. It can access the modules, and your application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



The application connects itself to the virtual hub to gain access to the module

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

¹ www.yoctopuce.com/EN/virtualhub.php



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

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

Language	Native	Native with DLL	Virtual hub
C++	•	•	•
Objective-C	•	-	•
Delphi	-	•	•
Python	-	•	•
VisualBasic .Net	-	•	•
C# .Net	-	•	•
Javascript	-	-	•
Node.js	-	-	•
PHP	-	-	•
Java	-	-	•
Java for Android	•	-	•
Command line	•	-	•

Support methods for different languages

Limitations of the Yoctopuce libraries

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

5.7. Programming, where to start?

At this point of the user's guide, you should know the main theoretical points of your Yocto-4-20mA-Rx. 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-4-20mA-Rx.

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

6. Using the Yocto-4-20mA-Rx in command line

When you want to perform a punctual operation on your Yocto-4-20mA-Rx, 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-4-20mA-Rx, open a shell, and start working by typing for example:

```
C:\>YGenericSensor 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 GenericSensor function

To control the GenericSensor function of your Yocto-4-20mA-Rx, you need the YGenericSensor executable file.

For instance, you can launch:

```
C:\>YGenericSensor any get_currentValue
```

This example uses the "any" target to indicate that we want to work on the first GenericSensor 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-4-20mA-Rx module with the *RX420MA1-123456* serial number which you have called "MyModule", and its genericSensor1 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:\>YGenericSensor RX420MA1-123456.genericSensor1 describe
C:\>YGenericSensor RX420MA1-123456.MyFunction describe
C:\>YGenericSensor MyModule.genericSensor1 describe
C:\>YGenericSensor MyModule.MyFunction describe
C:\>YGenericSensor MyFunction describe
```

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

```
C:\>YGenericSensor all describe
```

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

```
C:\>YGenericSensor /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 RX420MA1-12346 set_logicalName MonPremierModule
C:\>YModule RX420MA1-12346 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 RX420MA1-12346 set_logicalName MonPremierModule
C:\>YModule RX420MA1-12346 saveToFlash
```

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

```
C:\>YModule -s RX420MA1-12346 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-4-20mA-Rx with Javascript

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

At the time of writing, the Javascript library functions with any recent browser ... except Opera. It is likely that Opera will end up working with the Yoctopuce library one of these days¹, but it is not the case right now.

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

7.1. Getting ready

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

- The Javascript 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, connect your modules, run the VirtualHub software, and you are ready to start your first tests. You do not need to install any driver.

7.2. Control of the GenericSensor function

A few lines of code are enough to use a Yocto-4-20mA-Rx. Here is the skeleton of a JavaScript code snippet to use the GenericSensor function.

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

// Get access to your device, through the VirtualHub running locally
yRegisterHub('http://127.0.0.1:4444/');
var genericsensor = yFindGenericSensor("RX420MA1-123456.genericSensor1");

// Check that the module is online to handle hot-plug
if(genericsensor.isOnline())
```

¹ Actually, as soon as Opera implements support for the HTTP Access-Control-Allow-Origin header.

² www.yoctopuce.com/EN/libraries.php

³ www.yoctopuce.com/EN/virtualhub.php

```
{
  // Use genericsensor.get_currentValue(), ...
}
```

Let us look at these lines in more details.

yocto_api.js and yocto_genericsensor.js

These two Javascript includes provide access to functions allowing you to manage Yoctopuce modules. `yocto_api.js` must always be included, `yocto_genericsensor.js` is necessary to manage modules containing a generic sensor, such as Yocto-4-20mA-Rx.

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.

yFindGenericSensor

The `yFindGenericSensor` function allows you to find a generic 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-4-20mA-Rx module with serial number `RX420MA1-123456` which you have named `"MyModule"`, and for which you have given the `genericSensor1` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
var genericsensor = yFindGenericSensor("RX420MA1-123456.genericSensor1");
var genericsensor = yFindGenericSensor("RX420MA1-123456.MyFunction");
var genericsensor = yFindGenericSensor("MyModule.genericSensor1");
var genericsensor = yFindGenericSensor("MyModule.MyFunction");
var genericsensor = yFindGenericSensor("MyFunction");
```

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

isOnline

The `isOnline()` method of the object returned by `yFindGenericSensor` 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 `yFindGenericSensor` provides the current currently measured by the Yocto-4-20mA-Rx. The value returned is a floating number, converted to the physical value measured by the 4..20mA sensor.

A real example

Open your preferred text editor⁴, copy the code sample below, save it in the same directory as the Yoctopuce library files and then use your preferred web browser to access this page. The code is also provided in the directory **Examples/Doc-GettingStarted-Yocto-4-20mA-Rx** of the Yoctopuce library.

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

The example is coded to be used either from a web server, or directly by opening the file on the local machine. Note that this latest solution does not work with some versions of Internet Explorer, in particular IE 9 on Windows 7, which is not able to open network connections when working on a local

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

file. In order to use Internet Explorer, you should load the example from a web server. No such problem exists with Chrome, Firefox or Safari.

If your Yocto-4-20mA-Rx 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-4-20mA-Rx is connected and where you run the VirtualHub.

```
<HTML>
<HEAD>
<TITLE> Hello World</TITLE>
<SCRIPT type="text/javascript" src="yocto_api.js"></SCRIPT>
<SCRIPT type="text/javascript" src="yocto_genericSensor.js"></SCRIPT>
<SCRIPT language='javascript1.5' type='text/JavaScript'>
<!--

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

function refresh()
{
    var sensor, serial = document.getElementById('serial').value;

    if(serial == '') {
        // or use any connected module suitable for the demo
        sensor = yFirstGenericSensor();
        if(sensor) {
            serial = sensor.module().get_serialNumber();
            document.getElementById('serial').value = serial;
        }
    }

    sensor1 = yFindGenericSensor(serial+".genericSensor1");
    sensor2 = yFindGenericSensor(serial+".genericSensor2");

    if ((sensor1.isOnline()) && (sensor2.isOnline())) {
        document.getElementById('msg').value = '';
        document.getElementById("sensor-val1").value = sensor1.get_currentValue() +
sensor1.get_unit() ;
        document.getElementById("sensor-val2").value = sensor2.get_currentValue() +
sensor2.get_unit() ;
    } else {
        document.getElementById('msg').value = 'Module not connected';
    }
    setTimeout('refresh()',500);
}
-->
</SCRIPT>
</HEAD>
<BODY onload='refresh();'>
Module to use: <input id='serial'>
<input id='msg' style='color:red;border:none;' readonly><br>
channel 1 : <input id='sensor-val1' readonly><br>
channel 2 : <input id='sensor-val2' readonly><br>
</BODY>
</HTML>
```

7.3. Control of the module part

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

```
<HTML>
<HEAD>
<TITLE>Module Control</TITLE>
<SCRIPT type="text/javascript" src="yocto_api.js"></SCRIPT>
<SCRIPT language='javascript1.5' type='text/JavaScript'>
<!--

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

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

var module;

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

    module = yFindModule(serial);
    if(module.isOnline()) {
        document.getElementById('msg').value = '';
        var html = 'serial: '+module.get_serialNumber()+'<br>';
        html += 'logical name: '+module.get_logicalName()+'<br>';
        html += 'luminosity: '+module.get_luminosity()+'%<br>';
        html += 'beacon: ';
        if (module.get_beacon() == Y_BEACON_ON)
            html += "ON <a href='javascript:beacon(Y_BEACON_OFF)'>switch off</a><br>";
        else
            html += "OFF <a href='javascript:beacon(Y_BEACON_ON)'>switch on</a><br>";

        html += 'upTime: '+parseInt(module.get_upTime()/1000)+' sec<br>';
        html += 'USB current: '+module.get_usbCurrent()+' mA<br>';
        html += 'logs:<br><pre>'+module.get_lastLogs()+'</pre><br>';
        document.getElementById('data').innerHTML = html;
    } else {
        document.getElementById('msg').value = 'Module not connected';
    }
    setTimeout('refresh()',1000);
}

function beacon(state)
{
    module.set_beacon(state);
    refresh();
}

-->
</SCRIPT>
</HEAD>
<BODY onload='refresh();'>
Module to use: <input id='serial'>
<input id='msg' style='color:red;border:none;' readonly><br>
<span id='data'></span>
</BODY>
</HTML>
```

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

Changing the module settings

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

```
<HTML>
<HEAD>
<TITLE>Change module settings</TITLE>
<SCRIPT type="text/javascript" src="yocto_api.js"></SCRIPT>
```



```

<SCRIPT language='javascript1.5' type='text/JavaScript'>
<!--
// Use explicit error handling rather than exceptions
yDisableExceptions();

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

var module;

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

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

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

```

Warning: the number of write cycles of the nonvolatile memory of the module is limited. When this limit is reached, nothing 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>Modules inventory</TITLE>
<SCRIPT type="text/javascript" src="yocto_api.js"></SCRIPT>
<SCRIPT language='javascript1.5' type='text/JavaScript'>

```

```

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

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

function refresh()
{
    yUpdateDeviceList();

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

```

7.4. Error handling

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

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

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

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

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

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

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

8. Using Yocto-4-20mA-Rx 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 GenericSensor function

A few lines of code are enough to use a Yocto-4-20mA-Rx. Here is the skeleton of a PHP code snippet to use the GenericSensor function.

```
include('yocto_api.php');  
include('yocto_genericsensor.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);
$genericsensor = yFindGenericSensor("RX420MA1-123456.genericSensor1");

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

Let's look at these lines in more details.

yocto_api.php and yocto_genericsensor.php

These two PHP includes provides access to the functions allowing you to manage Yoctopuce modules. `yocto_api.php` must always be included, `yocto_genericsensor.php` is necessary to manage modules containing a generic sensor, such as Yocto-4-20mA-Rx.

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.

yFindGenericSensor

The `yFindGenericSensor` function allows you to find a generic 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-4-20mA-Rx module with serial number `RX420MA1-123456` which you have named `"MyModule"`, and for which you have given the `genericSensor1` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
$genericsensor = yFindGenericSensor("RX420MA1-123456.genericSensor1");
$genericsensor = yFindGenericSensor("RX420MA1-123456.MyFunction");
$genericsensor = yFindGenericSensor("MyModule.genericSensor1");
$genericsensor = yFindGenericSensor("MyModule.MyFunction");
$genericsensor = yFindGenericSensor("MyFunction");
```

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

isOnline

The `isOnline()` method of the object returned by `yFindGenericSensor` 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 `yFindGenericSensor` provides the current currently measured by the Yocto-4-20mA-Rx. The value returned is a floating number, converted to the physical value measured by the 4..20mA sensor.

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-4-20mA-Rx** 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_genericSensor.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
    $sensor = yFindGenericSensor("$serial.genericSensor1");
    if (!$sensor->isOnline()) {
        die("Module not connected (check serial and USB cable)");
    }
} else {
    // or use any connected module suitable for the demo
    $sensor = yFirstGenericSensor();
    if(is_null($sensor)) {
        die("No module connected (check USB cable)");
    } else {
        $serial = $sensor->module()->get_serialnumber();
    }
}
Print("Module to use: <input name='serial' value='$serial'><br>");

$sensor1 = yFindGenericSensor("$serial.genericSensor1");
Printf("GenericSensor channel 1: %.1f %s<br>", $sensor1->get_currentValue(), $sensor1->get_unit());

$sensor2 = yFindGenericSensor("$serial.genericSensor2");
Printf("GenericSensor channel 2: %.1f %s<br>", $sensor2->get_currentValue(), $sensor2->get_unit());

// 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());
?>
<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_xxxx()` method. For more details regarding the used functions, refer to the API chapters.

Changing the module settings

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

```

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

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

    // Setup the API to use the VirtualHub on local machine

```



```

if(yRegisterHub('http://127.0.0.1:4444/', $errmsg) != YAPI_SUCCESS) {
    die("Cannot contact VirtualHub on 127.0.0.1");
}

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

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

```

Warning: the number of write cycles of the nonvolatile memory of the module is limited. When this limit is reached, nothing 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();
    }
?>
</TT>
</BODY>
</HTML>

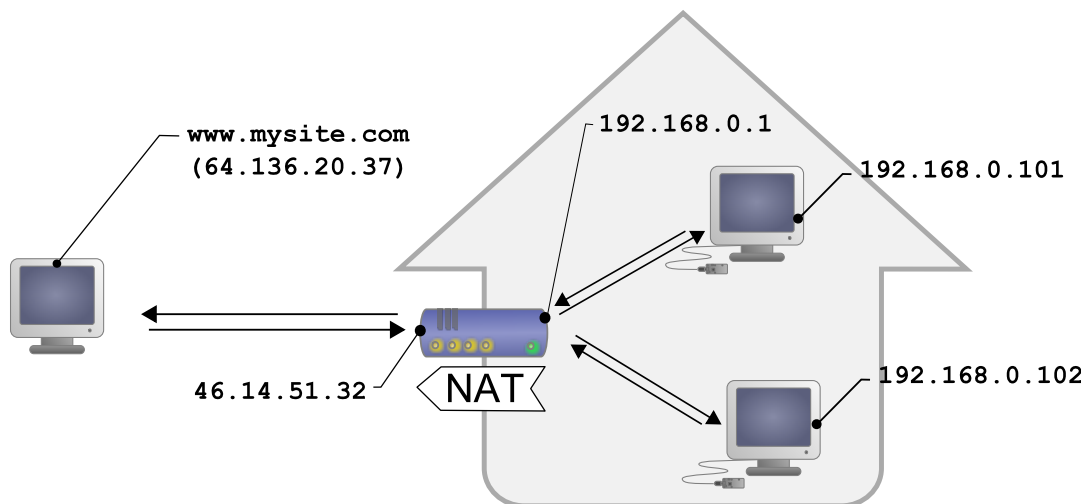
```

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.

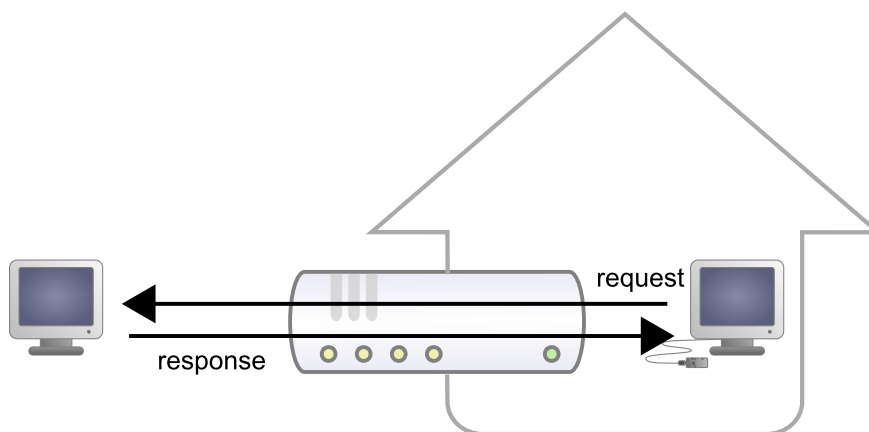
The NAT filter: advantages and disadvantages

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.

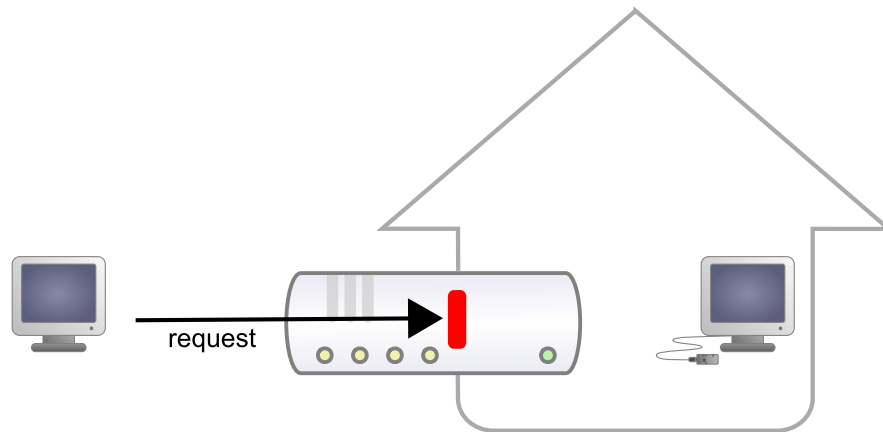


Typical DSL configuration: LAN machines are isolated from the outside by the DSL router

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.



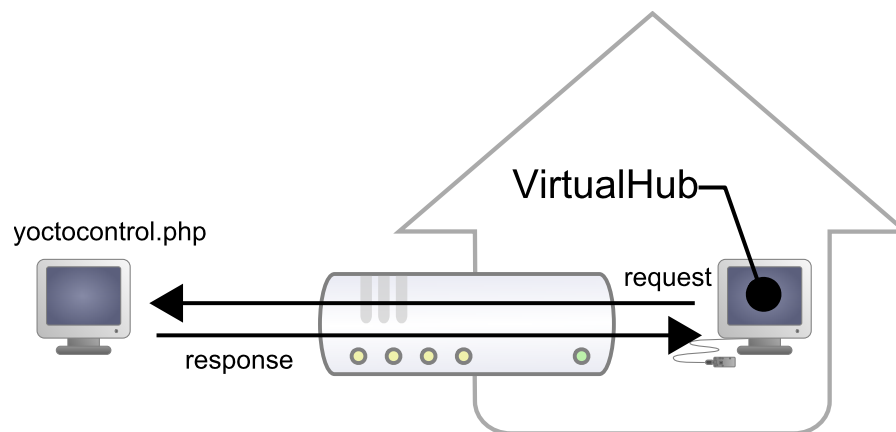
Responses from request from LAN machines are routed.



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	configure view log file
RELAYHI1-00055		Yocto-PowerRelay	configure view log file beacon
TMPSENS1-05E7F		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

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:

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

bounds for the returned value. In the case of functions which do not normally return information, the return value is `YAPI_SUCCESS` if everything went well, and a different error code in case of failure.

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

9. Using Yocto-4-20mA-Rx 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 GenericSensor function

A few lines of code are enough to use a Yocto-4-20mA-Rx. Here is the skeleton of a C++ code snippet to use the `GenericSensor` function.

```
#include "yocto_api.h"
#include "yocto_genericsensor.h"

[...]
String errmsg;
YGenericSensor *genericsensor;

// Get access to your device, connected locally on USB for instance
yRegisterHub("usb", errmsg);
genericsensor = yFindGenericSensor("RX420MA1-123456.genericSensor1");
```

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

Let's look at these lines in more details.

yocto_api.h et yocto_genericsensor.h

These two include files provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api.h` must always be used, `yocto_genericsensor.h` is necessary to manage modules containing a generic sensor, such as Yocto-4-20mA-Rx.

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.

yFindGenericSensor

The `yFindGenericSensor` function allows you to find a generic 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-4-20mA-Rx module with serial number `RX420MA1-123456` which you have named `"MyModule"`, and for which you have given the `genericSensor1` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
YGenericSensor *genericsensor = yFindGenericSensor("RX420MA1-123456.genericSensor1");
YGenericSensor *genericsensor = yFindGenericSensor("RX420MA1-123456.MyFunction");
YGenericSensor *genericsensor = yFindGenericSensor("MyModule.genericSensor1");
YGenericSensor *genericsensor = yFindGenericSensor("MyModule.MyFunction");
YGenericSensor *genericsensor = yFindGenericSensor("MyFunction");
```

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

isOnline

The `isOnline()` method of the object returned by `yFindGenericSensor` 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 `yFindGenericSensor` provides the current currently measured by the Yocto-4-20mA-Rx. The value returned is a floating number, converted to the physical value measured by the 4..20mA sensor.

A real example

Launch your C++ environment and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-4-20mA-Rx** 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_genericsensor.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          (use any discovered device)" << endl;
    u64 now = yGetTickCount(); // dirty active wait loop
    while (yGetTickCount()-now<3000);
    exit(1);
}

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

    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") {
        sensor = yFirstGenericSensor();
        if (sensor==NULL) {
            cout << "No module connected (check USB cable)" << endl;
            return 1;
        }
    } else {
        sensor = yFindGenericSensor(target + ".temperature1");
    }

    YGenericSensor *s1 = yFindGenericSensor(sensor->get_module()->get_serialNumber() +
".genericSensor1");
    YGenericSensor *s2 = yFindGenericSensor(sensor->get_module()->get_serialNumber() +
".genericSensor2");

    string unitSensor1,unitSensor2;

    if (s1->isOnline()) unitSensor1 = s1->get_unit();
    if (s2->isOnline()) unitSensor2 = s2->get_unit();

    while (s1->isOnline() && s2->isOnline()) {
        double value =s1->get_currentValue();
        cout << "Channel 1 : " << s1->get_currentValue() << unitSensor1;
        value =s2->get_currentValue();
        cout << "Channel 2 : " << s2->get_currentValue() << unitSensor2;
        cout << " (press Ctrl-C to exit)" << endl;
        ySleep(1000,errmsg);
    };

    cout << "Module not connected (check identification and USB cable)";
    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;
    }
    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      errormsg;

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

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

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

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

```

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

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

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

    YModule *module = yFirstModule();
    while (module != NULL) {
        cout << module->get_serialNumber() << " ";
        cout << module->get_productName() << endl;
        module = module->nextModule();
    }
    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 `yDisableExceptions()` function to commute the library to a mode where exceptions for all the functions are systematically replaced by specific return values, which can be tested by the caller when necessary. For each function, the name of each return value in case of error is systematically documented in the library reference. The name always follows the same logic: a `getState()` method returns a `Y_STATE_INVALID` value, a `getCurrentValue` 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-4-20mA-Rx 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 GenericSensor function

Launch Xcode 4.2 and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-4-20mA-Rx** of the Yoctopuce library.

```
#import <Foundation/Foundation.h>
#import "yocto_api.h"
#import "yocto_genericsensor.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;
YGenericSensor *sensor,*sensor1,*sensor2;

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

@autoreleasepool {
    NSString *target = [NSString stringWithUTF8String:argv[1]];

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

    if ([target isEqualToString:@"any"]) {
        // retrieve any generic sensor
        sensor = [YGenericSensor FirstGenericSensor];
        if (sensor==NULL) {
            NSLog(@"No module connected (check USB cable)");
            return 1;
        }
    } else {
        sensor = [YGenericSensor FindGenericSensor:target];
    }

    // we need to retrieve both DC and AC current from the device.
    if (![sensor isOnline]) {
        NSLog(@"No module connected (check USB cable)");
        return 1;
    }
    YModule *m = [sensor module];
    sensor1 = [YGenericSensor FindGenericSensor:[m.serialNumber
stringByAppendingString:@"genericSensor1"]];
    sensor2 = [YGenericSensor FindGenericSensor:[m.serialNumber
stringByAppendingString:@"genericSensor2"]];

    while([m isOnline]) {
        NSLog(@"Channel 1 : %f %@",[sensor1 currentValue],[sensor1 get_unit]);
        NSLog(@"Channel 2 : %f %@",[sensor2 currentValue],[sensor2 get_unit]);
        NSLog(@" (press Ctrl-C to exit)");
        [YAPI Sleep:1000:NULL];
    }
    NSLog(@"Module not connected (check identification and USB cable)");
}
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_genericsensor.h

These two import files provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api.h` must always be used, `yocto_genericsensor.h` is necessary to manage modules containing a generic sensor, such as Yocto-4-20mA-Rx.

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.

yFindGenericSensor

The `yFindGenericSensor` function allows you to find a generic 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-4-20mA-Rx module with serial number `RX420MA1-123456` which you have named `"MyModule"`, and for which you have given the `genericSensor1` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.


```
YGenericSensor *genericsensor = yFindGenericSensor(@"RX420MA1-123456.genericSensor1");
YGenericSensor *genericsensor = yFindGenericSensor(@"RX420MA1-123456.MyFunction");
YGenericSensor *genericsensor = yFindGenericSensor(@"MyModule.genericSensor1");
YGenericSensor *genericsensor = yFindGenericSensor(@"MyModule.MyFunction");
YGenericSensor *genericsensor = yFindGenericSensor(@"MyFunction");
```

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

isOnline

The `isOnline()` method of the object returned by `yFindGenericSensor` 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 `yFindGenericSensor` provides the current currently measured by the Yocto-4-20mA-Rx. The value returned is a floating number, converted to the physical value measured by the 4..20mA sensor.

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]];
        YModule *module = [YModule FindModule:serial_or_name]; // use serial or logical
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:      %d 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
);
        }
    }
}
```

```

    }
    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(yRegisterHub(@"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]];
        YModule *module = yFindModule(serial_or_name); // use serial or logical name

        if (module.isOnline) {
            if (argc >= 3){
                NSString *newname = [NSString stringWithUTF8String:argv[2]];
                if (!yCheckLogicalName(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);
        }
    }
    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(yRegisterHub(@"usb", &error) != YAPI_SUCCESS) {
            NSLog(@"RegisterHub error: %@\n", [error localizedDescription]);
            return 1;
        }

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

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

10.3. Error handling

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

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

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

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

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

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

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

11. Using Yocto-4-20mA-Rx 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 GenericSensor function

A few lines of code are enough to use a Yocto-4-20mA-Rx. Here is the skeleton of a Visual Basic code snippet to use the `GenericSensor` function.

```
[...]
Dim errmsg As String
Dim genericsensor As YGenericSensor

REM Get access to your device, connected locally on USB for instance
yRegisterHub("usb", errmsg)
genericsensor = yFindGenericSensor("RX420MA1-123456.genericSensor1")

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

yFindGenericSensor

The `yFindGenericSensor` function allows you to find a generic 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-4-20mA-Rx module with serial number `RX420MA1-123456` which you have named `"MyModule"`, and for which you have given the `genericSensor1` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
genericsensor = yFindGenericSensor("RX420MA1-123456.genericSensor1")
genericsensor = yFindGenericSensor("RX420MA1-123456.MyFunction")
genericsensor = yFindGenericSensor("MyModule.genericSensor1")
genericsensor = yFindGenericSensor("MyModule.MyFunction")
genericsensor = yFindGenericSensor("MyFunction")
```

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

isOnline

The `isOnline()` method of the object returned by `yFindGenericSensor` 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 `yFindGenericSensor` provides the current currently measured by the Yocto-4-20mA-Rx. The value returned is a floating number, converted to the physical value measured by the 4..20mA sensor.

A real example

Launch Microsoft VisualBasic and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-4-20mA-Rx** 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

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

        Dim sensor, ch1, ch2 As YGenericSensor

        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
            sensor = yFirstGenericSensor()
            If sensor Is Nothing Then
                Console.WriteLine("No module connected (check USB cable) ")
            End If
            Console.WriteLine("using: " + sensor.get_module().get_serialNumber())
        Else
            sensor = yFindGenericSensor(target + ".genericSensor1")

        End If

        REM retrieve module serial number
        serial = sensor.get_module().get_serialNumber()

        REM retrieve both channels
        ch1 = yFindGenericSensor(serial + ".genericSensor1")
        ch2 = yFindGenericSensor(serial + ".genericSensor2")

        While (ch1.isOnline() And ch2.isOnline())
            Console.WriteLine("channel 1: " + Str(ch1.get_currentValue()) + ch1.get_unit())
            Console.WriteLine("channel 2: " + Str(ch2.get_currentValue()) + ch2.get_unit())
            Console.WriteLine(" (press Ctrl-C to exit)")
            ySleep(1000, errmsg)
        End While
        Console.WriteLine("Module not connected (check identification and USB cable)")

    End Sub

End Module
```

```
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 Sub
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 If
End If

If argv.Length < 2 Then usage()

m = yFindModule(argv(1)) REM use serial or logical name

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

End Sub

End 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

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

End Sub
```

```

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

End Sub

End Module

```

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

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

12. Using Yocto-4-20mA-Rx 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.

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.

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

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

² www.yoctopuce.com/EN/libraries.php

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

Then add in the same manner the `yapi.dll` DLL, located in the `Sources/dll` directory⁴. Then, from the **Solution Explorer** window, right click on the DLL, select **Properties** and in the **Properties** panel, set the **Copy to output folder** to **always**. You are now ready to use your Yoctopuce modules from Visual Studio.

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

12.3. Control of the GenericSensor function

A few lines of code are enough to use a Yocto-4-20mA-Rx. Here is the skeleton of a C# code snippet to use the `GenericSensor` function.

```
[...]
string errmsg = "";
YGenericSensor genericsensor;

// Get access to your device, connected locally on USB for instance
YAPI.RegisterHub("usb", errmsg);
genericsensor = YGenericSensor.FindGenericSensor("RX420MA1-123456.genericSensor1");

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

YGenericSensor.FindGenericSensor

The `YGenericSensor.FindGenericSensor` function allows you to find a generic 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-4-20mA-Rx module with serial number `RX420MA1-123456` which you have named `"MyModule"`, and for which you have given the `genericSensor1` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
genericsensor = YGenericSensor.FindGenericSensor("RX420MA1-123456.genericSensor1");
genericsensor = YGenericSensor.FindGenericSensor("RX420MA1-123456.MyFunction");
genericsensor = YGenericSensor.FindGenericSensor("MyModule.genericSensor1");
genericsensor = YGenericSensor.FindGenericSensor("MyModule.MyFunction");
genericsensor = YGenericSensor.FindGenericSensor("MyFunction");
```

`YGenericSensor.FindGenericSensor` returns an object which you can then use at will to control the generic sensor.

isOnline

The `isOnline()` method of the object returned by `YGenericSensor.FindGenericSensor` 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 `GenericSensor.FindGenericSensor` provides the current currently measured by the Yocto-4-20mA-Rx. The value returned is a floating number, converted to the physical value measured by the 4..20mA sensor.

A real example

Launch Microsoft Visual C# and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-4-20mA-Rx** of the Yoctopuce library.

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

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

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

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

            YGenericSensor tsensor;

            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")
            {
                tsensor = YGenericSensor.FirstGenericSensor();

                if (tsensor == null)
                {
                    Console.WriteLine("No module connected (check USB cable) ");
                    Environment.Exit(0);
                }
                Console.WriteLine("Using: " + tsensor.get_module().get_serialNumber());
            }
            else
            {
                tsensor = YGenericSensor.FindGenericSensor(target + ".genericSensor1");
            }

            // retrieve module serial
            string serial = tsensor.get_module().get_serialNumber();

            // retrieve both channels
            YGenericSensor ch1 = YGenericSensor.FindGenericSensor(serial + ".genericSensor1");
            YGenericSensor ch2 = YGenericSensor.FindGenericSensor(serial + ".genericSensor2");

            string unitSensor1="", unitSensor2="";
            if (ch1.isOnline()) unitSensor1 =ch1.get_unit();
        }
    }
}
```

```

        if (ch2.isOnline()) unitSensor2 = ch2.get_unit();

        while (ch1.isOnline() && ch2.isOnline())
        { Console.WriteLine("Channel 1 : " + ch1.get_currentValue().ToString() + unitSensor1);
          Console.WriteLine("    Channel 2 : " + ch2.get_currentValue().ToString() + unitSensor2);
          Console.WriteLine("    (press Ctrl-C to exit)");
          YAPI.Sleep(1000, ref errmsg);
        }

        Console.WriteLine("Module not connected (check identification and USB cable)");
    }
}
}

```

12.4. Control of the module part

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

```

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

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

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

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

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

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

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

                Console.WriteLine("serial:          " + m.get_serialNumber());
                Console.WriteLine("logical name:  " + m.get_logicalName());
                Console.WriteLine("luminosity:    " + m.get_luminosity().ToString());
                Console.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)");
    }
}
}

```

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

Changing the module settings

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

```

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

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

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

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

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

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

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

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

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

```

}

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

Listing the modules

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

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

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

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

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

12.5. Error handling

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

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

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

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

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

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

13. Using Yocto-4-20mA-Rx 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.

13.1. 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.

13.2. Control of the GenericSensor 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_genericsensor;
```

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');
  sleep(3000);
  halt;
End;

var
  sensor,ch1,ch2 : TYGenericSensor;
  module : TYModule;
  errmsg,serial : string;
  unitSensor1,unitSensor2:string;

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 := yFirstGenericSensor();
    if sensor=nil then
    begin
      writeln('No module connected (check USB cable)');
      halt;
    end
  end
  else
    sensor:= YFindGenericSensor(paramstr(1)+'.genericSensor1');

  module:=sensor.get_module();
  serial:=module.get_serialNumber();
  ch1:=YFindGenericSensor(serial+'.genericSensor1');
  ch2:=YFindGenericSensor(serial+'.genericSensor2');

  if ch1.isOnline() then unitSensor1:= ch1.get_unit();
  if ch2.isOnline() then unitSensor2:= ch2.get_unit();

  while ch1.isOnline() and ch2.isOnline() do
  begin
    Write('Channel 1: '+FloatToStr(ch1.get_currentValue()+unitSensor1);
    Write('Channel 2: '+FloatToStr(ch2.get_currentValue()+unitSensor2);
    Writeln(' (press Ctrl-C to exit)');
    Sleep(1000);
  end;

  Writeln('Module not connected (check identification and USB cable)');
end.

```

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

yocto_api and yocto_genericsensor

These two units provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api` must always be used, `yocto_genericsensor` is necessary to manage modules containing a generic sensor, such as Yocto-4-20mA-Rx.

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.

yFindGenericSensor

The `yFindGenericSensor` function allows you to find a generic 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-4-20mA-Rx module with serial number *RX420MA1-123456* which you have named *"MyModule"*, and for which you have given the *genericSensor1* function the name *"MyFunction"*. The following five calls are strictly equivalent, as long as *"MyFunction"* is defined only once.

```
genericsensor := yFindGenericSensor("RX420MA1-123456.genericSensor1");
genericsensor := yFindGenericSensor("RX420MA1-123456.MyFunction");
genericsensor := yFindGenericSensor("MyModule.genericSensor1");
genericsensor := yFindGenericSensor("MyModule.MyFunction");
genericsensor := yFindGenericSensor("MyFunction");
```

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

isOnline

The `isOnline()` method of the object returned by `yFindGenericSensor` 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 `yFindGenericSensor` provides the current currently measured by the Yocto-4-20mA-Rx. The value returned is a floating number, converted to the physical value measured by the 4..20mA sensor.

13.3. 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 = 'RX420MA1-123456'; // 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
end
```

```

    else Writeln('Module not connected (check identification and USB cable)');
end;

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

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

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

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

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

```

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

Changing the module settings

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

```

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

const
    serial = 'RX420MA1-123456'; // use serial number or logical name

var
    module : TModule;
    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();

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

```

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

Listing the modules

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

```

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

var
    module : TYModule;
    errmsg : string;

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

    Writeln('Device list');

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

```

13.4. Error handling

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

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

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

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

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

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

14. Using the Yocto-4-20mA-Rx 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.

14.1. 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.

14.2. 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.

14.3. Control of the GenericSensor function

A few lines of code are enough to use a Yocto-4-20mA-Rx. Here is the skeleton of a Python code snippet to use the GenericSensor 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)
genericsensor = YGenericSensor.FindGenericSensor("RX420MA1-123456.genericSensor1")

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

YGenericSensor.FindGenericSensor

The `YGenericSensor.FindGenericSensor` function allows you to find a generic 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-4-20mA-Rx module with serial number *RX420MA1-123456* which you have named "*MyModule*", and for which you have given the *genericSensor1* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
genericsensor = YGenericSensor.FindGenericSensor("RX420MA1-123456.genericSensor1")
genericsensor = YGenericSensor.FindGenericSensor("RX420MA1-123456.MyFunction")
genericsensor = YGenericSensor.FindGenericSensor("MyModule.genericSensor1")
genericsensor = YGenericSensor.FindGenericSensor("MyModule.MyFunction")
genericsensor = YGenericSensor.FindGenericSensor("MyFunction")
```

`YGenericSensor.FindGenericSensor` returns an object which you can then use at will to control the generic sensor.

isOnline

The `isOnline()` method of the object returned by `YGenericSensor.FindGenericSensor` 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 `GenericSensor.FindGenericSensor` provides the current currently measured by the Yocto-4-20mA-Rx. The value returned is a floating number, converted to the physical value measured by the 4..20mA sensor.

A real example

Launch Python and open the corresponding sample script provided in the directory **Examples/Doc-GettingStarted-Yocto-4-20mA-Rx** 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_genericsensor 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 genericSensor sensor
    sensor = YGenericSensor.FirstGenericSensor()
    if sensor is None :
        die('No module connected')
else:
    sensor= YGenericSensor.FindGenericSensor(target + '.genericSensor1')

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

# retrieve module serial
serial=sensor.get_module().get_serialNumber()

# retrieve both channels
channel1 = YGenericSensor.FindGenericSensor(serial + '.genericSensor1')
channel2 = YGenericSensor.FindGenericSensor(serial + '.genericSensor2')

while channel1.isOnline() and channel2.isOnline():
    print("channel 1:  %f %s" % (channel1.get_currentValue(), channel1.get_unit()))
    print("channel 2:  %f %s" % (channel2.get_currentValue(), channel2.get_unit()))
    print("          (Ctrl-C to stop)")
    YAPI.Sleep(1000)

```

14.4. Control of the module part

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

```

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

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

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

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

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

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

```

```

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

```

Each property `xxx` of the module can be read thanks to a method of type `YModule.get_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.

```

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

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

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

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

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

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

```

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

14.5. Error handling

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

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

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

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

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

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

15. Using the Yocto-4-20mA-Rx 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.

15.1. 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.

15.2. Control of the GenericSensor function

A few lines of code are enough to use a Yocto-4-20mA-Rx. Here is the skeleton of a Java code snippet to use the GenericSensor function.

```
[...]  
  
// Get access to your device, connected locally on USB for instance  
YAPI.RegisterHub("127.0.0.1");  
genericsensor = YGenericSensor.FindGenericSensor("RX420MA1-123456.genericSensor1");  
  
// Hot-plug is easy: just check that the device is online  
if (genericsensor.isOnline())  
{ //Use genericsensor.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.

YGenericSensor.FindGenericSensor

The `YGenericSensor.FindGenericSensor` function allows you to find a generic 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-4-20mA-Rx module with serial number *RX420MA1-123456* which you have named *"MyModule"*, and for which you have given the *genericSensor1* function the name *"MyFunction"*. The following five calls are strictly equivalent, as long as *"MyFunction"* is defined only once.

```
genericSensor = YGenericSensor.FindGenericSensor("RX420MA1-123456.genericSensor1")
genericSensor = YGenericSensor.FindGenericSensor("RX420MA1-123456.MyFunction")
genericSensor = YGenericSensor.FindGenericSensor("MyModule.genericSensor1")
genericSensor = YGenericSensor.FindGenericSensor("MyModule.MyFunction")
genericSensor = YGenericSensor.FindGenericSensor("MyFunction")
```

`YGenericSensor.FindGenericSensor` returns an object which you can then use at will to control the generic sensor.

isOnline

The `isOnline()` method of the object returned by `YGenericSensor.FindGenericSensor` 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 `GenericSensor.FindGenericSensor` provides the current currently measured by the Yocto-4-20mA-Rx. The value returned is a floating number, converted to the physical value measured by the 4..20mA sensor.

A real example

Launch your Java environment and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-4-20mA-Rx** 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);
        }

        YGenericSensor sensor;
        if (args.length > 0) {
            sensor = YGenericSensor.FindGenericSensor(args[0]);
```



```

    } else {
        sensor = YGenericSensor.FirstGenericSensor();
    }
    if (sensor == null) {
        System.out.println("No module connected (check USB cable)");
        System.exit(1);
    }

    try {
        YGenericSensor s1 = YGenericSensor.FindGenericSensor(sensor.get_module(
).get_serialNumber() + ".genericSensor1");
        YGenericSensor s2 = YGenericSensor.FindGenericSensor(sensor.get_module(
).get_serialNumber() + ".genericSensor2");

        while (s1.isOnline() && s2.isOnline()) {
            double value = s1.get_currentValue();
            System.out.println("Channel 1 : " + s1.get_currentValue() + " " +
s1.get_unit());
            value = s2.get_currentValue();
            System.out.println("Channel 2 : " + s2.get_currentValue() + " " +
s2.get_unit());
            System.out.println("  (press Ctrl-C to exit)");
            YAPI.Sleep(1000);
        }

        } catch (YAPI_Exception ex) {
            System.out.println("Module " + sensor.describe() + " disconnected (check
identification and USB cable)");
        }
        YAPI.FreeAPI();
    }
}

```

15.3. 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();
    }
}

```

15.4. 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.

16. Using the Yocto-4-20mA-Rx 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.

16.1. 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.

16.2. 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.

16.3. 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.

16.4. 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>
```

16.5. Control of the GenericSensor function

A few lines of code are enough to use a Yocto-4-20mA-Rx. Here is the skeleton of a Java code snippet to use the GenericSensor function.

```
[...]

// Retrieving the object representing the module (connected here locally by USB)
YAPI.EnableUSBHost(this);
YAPI.RegisterHub("usb");
genericsensor = YGenericSensor.FindGenericSensor("RX420MA1-123456.genericSensor1");

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

YGenericSensor.FindGenericSensor

The `YGenericSensor.FindGenericSensor` function allows you to find a generic 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-4-20mA-Rx module with serial number *RX420MA1-123456* which you have named "MyModule", and for which you have given the *genericSensor1* function the name "MyFunction". The following five calls are strictly equivalent, as long as "MyFunction" is defined only once.

```
genericsensor = YGenericSensor.FindGenericSensor("RX420MA1-123456.genericSensor1")
genericsensor = YGenericSensor.FindGenericSensor("RX420MA1-123456.MyFunction")
genericsensor = YGenericSensor.FindGenericSensor("MyModule.genericSensor1")
genericsensor = YGenericSensor.FindGenericSensor("MyModule.MyFunction")
genericsensor = YGenericSensor.FindGenericSensor("MyFunction")
```

`YGenericSensor.FindGenericSensor` returns an object which you can then use at will to control the generic sensor.

isOnline

The `isOnline()` method of the object returned by `YGenericSensor.FindGenericSensor` 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 `GenericSensor.FindGenericSensor` provides the current currently measured by the Yocto-4-20mA-Rx. The value returned is a floating number, converted to the physical value measured by the 4..20mA sensor.

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.YGenericSensor;
import com.yoctopuce.YoctoAPI.YModule;

public class GettingStarted_Yocto_4_20mA_Rx extends Activity implements
OnItemClickListener
{

    private ArrayAdapter<String> aa;
    private String serial = "";
    private Handler handler = null;
    private TextView mChannel1Field,mChannel2Field;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.gettingstarted_yocto_4_20ma_rx);
        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();
        mChannel1Field = (TextView) findViewById(R.id.channel1field);
        mChannel2Field = (TextView) findViewById(R.id.channel2field);

    }

    @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-4-20mA-Rx")) {
                    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) {
                YGenericSensor sensor1 = YGenericSensor.FindGenericSensor(serial +
".genericSensor1");
                try {
                    mChannel1Field.setText(String.format("%.1f %s", sensor1.getCurrentValue
(), sensor1.getUnit()));
                } catch (YAPI_Exception e) {
                    e.printStackTrace();
                }
                YGenericSensor sensor2 = YGenericSensor.FindGenericSensor(serial +
".genericSensor2");
                try {
                    mChannel2Field.setText(String.format("%.1f %s", sensor2.getCurrentValue
(), sensor2.getUnit()));
                } catch (YAPI_Exception e) {
                    e.printStackTrace();
                }
            }
            handler.postDelayed(this, 1000);
        }
    };
}

```

16.6. Control of the module part

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

```

package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.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.setOnItemSelectedListener(this);
    aa = new ArrayAdapter<String>(this, android.R.layout.simple_spinner_item);
    aa.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
    my_spin.setAdapter(aa);
}

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

    try {
        aa.clear();
        YAPI.EnableUSBHost(this);
        YAPI.RegisterHub("usb");
        YModule r = YModule.FirstModule();
        while (r != null) {
            String hwid = r.get_hardwareId();
            aa.add(hwid);
            r = r.nextModule();
        }
    } catch (YAPI_Exception e) {
        e.printStackTrace();
    }
    // refresh Spinner with detected relay
    aa.notifyDataSetChanged();
}

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

private void DisplayModuleInfo()
{
    TextView field;
    if (module == null)
        return;
    try {
        field = (TextView) findViewById(R.id.serialfield);
        field.setText(module.getSerialNumber());
        field = (TextView) findViewById(R.id.logicalnamefield);
        field.setText(module.getLogicalName());
        field = (TextView) findViewById(R.id.luminosityfield);
        field.setText(String.format("%d%", module.getLuminosity()));
        field = (TextView) findViewById(R.id.uptimefield);
        field.setText(module.getUpTime() / 1000 + " sec");
        field = (TextView) findViewById(R.id.usbcurrentfield);
        field.setText(module.getUsbCurrent() + " mA");
        Switch sw = (Switch) findViewById(R.id.beaconswitch);
        Log.d("switch", "beacon" + module.get_beacon());
        sw.setChecked(module.getBeacon() == YModule.BEACON_ON);
        field = (TextView) findViewById(R.id.logs);
        field.setText(module.get_lastLogs());

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

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

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

```

```

{
}

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

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

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

```

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

Changing the module settings

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

```

package com.yoctopuce.doc_examples;

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

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

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

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

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

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

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

    @Override
    protected void onStop()
    {
    }
}
```

```
{  
    super.onStop();  
    YAPI.FreeAPI();  
}  
  
}
```

16.7. 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.

17. Advanced programming

The preceding chapters have introduced, in each available language, the basic programming functions which can be used with your Yocto-4-20mA-Rx 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.

17.1. 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 generic 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(YGenericSensor 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 `YGenericSensor` 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 `GenericSensor` function with the help of the `registerValueCallback` method. The following example sets up a callback for the first available `GenericSensor` function.

```
YGenericSensor f = YGenericSensor.FirstGenericSensor();
f.registerValueCallback(genericSensor1ChangeCallBack)
```

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(YGenericSensor 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 `GenericSensor` 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 `GenericSensor` function.

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

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

17.2. The data logger

Your Yocto-4-20mA-Rx 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-4-20mA-Rx 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-4-20mA-Rx 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-4-20mA-Rx 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-4-20mA-Rx does not have a battery, it cannot guess alone the current time when powered on. Nevertheless, the Yocto-4-20mA-Rx 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-4-20mA-Rx 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-4-20mA-Rx 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-4-20mA-Rx 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-4-20mA-Rx 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-4-20mA-Rx 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-4-20mA-Rx 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-4-20mA-Rx might be subject to a clock skew (theoretically up to 0.3%).

17.3. Sensor calibration

Your Yocto-4-20mA-Rx 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-4-20mA-Rx. 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-4-20mA-Rx 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-4-20mA-Rx. On the other hand, the API supports an external arbitrary calibration, implemented with callbacks.

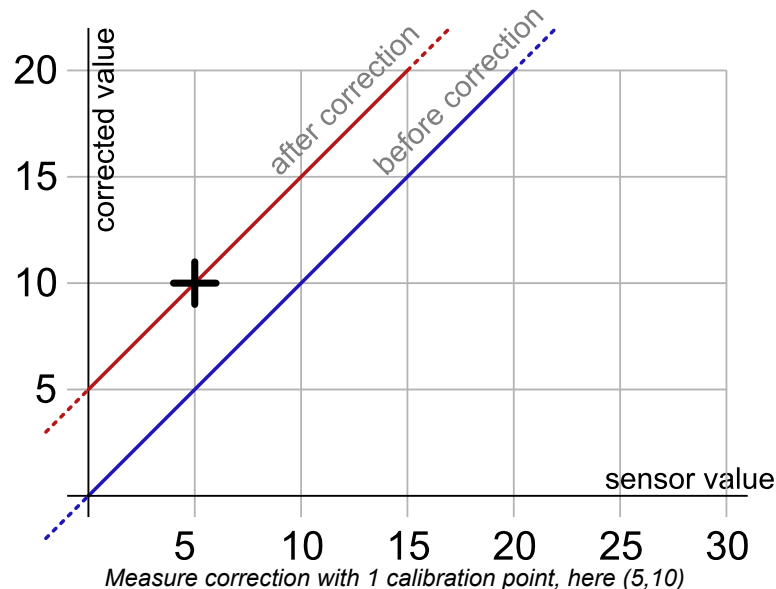
1 to 5 point linear interpolation

These transformations are performed directly inside the Yocto-4-20mA-Rx 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 `genericSensor1` function returns *b*.

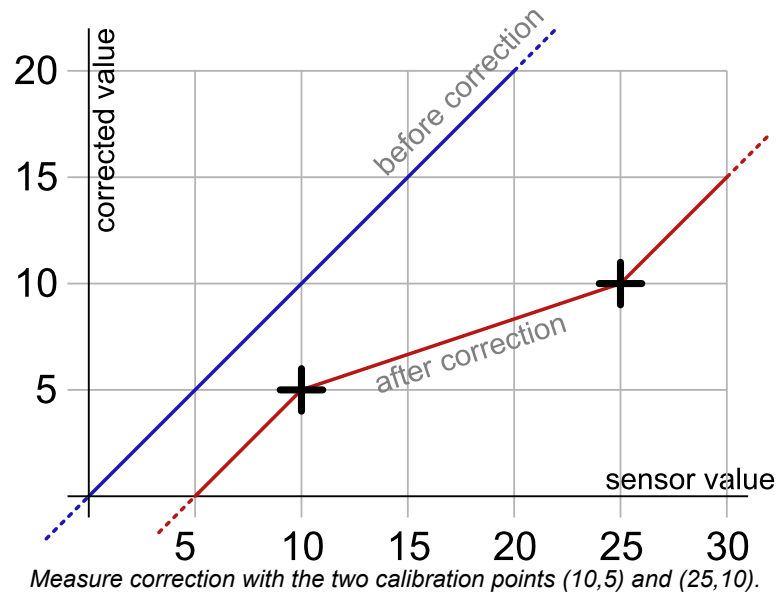


The application is very simple: you only need to call the `calibrateFromPoints()` method of the function you wish to correct. The following code applies the correction illustrated on the graph above to the first `genericSensor1` function found. Note the call to the `saveToFlash` method of the module hosting the function, so that the module does not forget the calibration as soon as it is disconnected.

```
Double[] ValuesBefore = {5};
Double[] ValuesAfter  = {10};
YGenericSensor f = YGenericSensor.FirstGenericSensor();
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 `genericSensor1` 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.



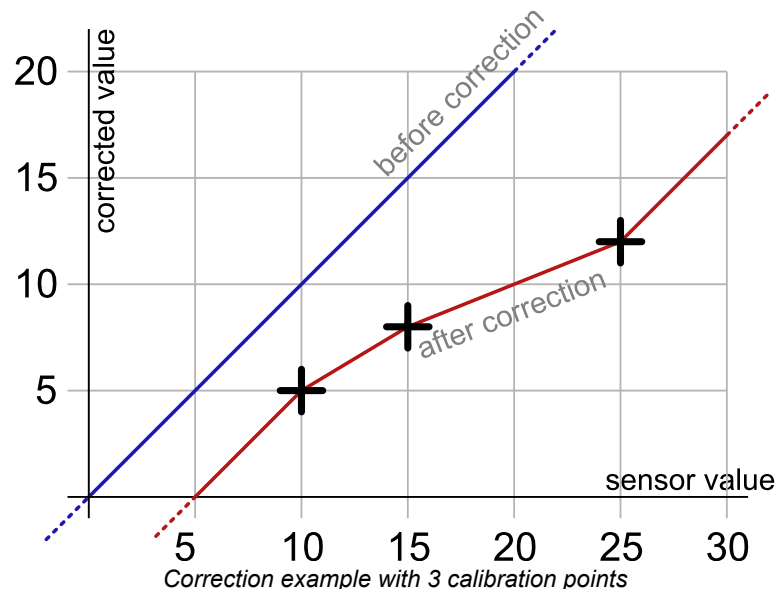
The code allowing you to program this calibration is very similar to the preceding code example.

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



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

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

Arbitrary interpolation

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

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

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

18. Using with unsupported languages

Yoctopuce modules can be driven from most common programming languages. New languages are regularly added, depending on the interest expressed by Yoctopuce product users. Nevertheless, some languages are not, and will never be, supported by Yoctopuce. There can be several reasons for this: compilers which are not available anymore, unadapted environments, etc.

However, there are alternative methods to access Yoctopuce modules from an unsupported programming language.

18.1. Command line

The easiest method to drive Yoctopuce modules from an unsupported programming language is to use the command line API through system calls. The command line API is in fact made of a group of small executables which are easy to call. Their output is also easy to analyze. As most programming languages allow you to make system calls, the issue is solved with a few lines of code.

However, if the command line API is the easiest solution, it is neither the fastest nor the most efficient. For each call, the executable must initialize its own API and make an inventory of USB connected modules. This requires about one second per call.

18.2. VirtualHub and HTTP GET

The *VirtualHub* is available on almost all current platforms. It is generally used as a gateway to provide access to Yoctopuce modules from languages which prevent direct access to hardware layers of a computer (JavaScript, PHP, Java, ...).

In fact, the *VirtualHub* is a small web server able to route HTTP requests to Yoctopuce modules. This means that if you can make an HTTP request from your programming language, you can drive Yoctopuce modules, even if this language is not officially supported.

REST interface

At a low level, the modules are driven through a REST API. Thus, to control a module, you only need to perform appropriate requests on the *VirtualHub*. By default, the *VirtualHub* HTTP port is 4444.

An important advantage of this technique is that preliminary tests are very easy to implement. You only need a *VirtualHub* and a simple web browser. If you copy the following URL in your preferred browser, while the *VirtualHub* is running, you obtain the list of the connected modules.

```
http://127.0.0.1:4444/api/services/whitePages.txt
```

Note that the result is displayed as text, but if you request *whitePages.xml*, you obtain an XML result. Likewise, *whitePages.json* allows you to obtain a JSON result. The *html* extension even allows you to display a rough interface where you can modify values in real time. The whole REST API is available in these different formats.

Driving a module through the REST interface

Each Yoctopuce module has its own REST interface, available in several variants. Let us imagine a Yocto-4-20mA-Rx with the *RX420MA1-12345* 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/RX420MA1-12345/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/RX420MA1-12345/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/RX420MA1-12345/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 *genericsensor* function, build the following URL:

```
http://127.0.0.1:4444/bySerial/RX420MA1-12345/api/genericsensor.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/RX420MA1-12345/api/genericsensor/logicalName
```

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

```
http://127.0.0.1:4444/bySerial/RX420MA1-12345/api/genericsensor?logicalName=myFunction
```

You can find the list of available attributes for your Yocto-4-20mA-Rx 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/RX420MA1-12345/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/RX420MA1-12345/dataLogger.json?
id=genericsensor&utc=1389801080
```

18.3. Using dynamic libraries

The low level Yoctopuce API is available under several formats of dynamic libraries written in C. The sources are available with the C++ API. If you use one of these low level libraries, you do not need the *VirtualHub* anymore.

Filename	Platform
libyapi.dylib	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      = 'RX420MA1-12345';
    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.

```

18.4. Porting the high level library

As all the sources of the Yoctopuce API are fully provided, you can very well port the whole API in the language of your choice. Note, however, that a large portion of the API source code is automatically generated.

Therefore, it is not necessary for you to port the complete API. You only need to port the *yocto_api* file and one file corresponding to a function, for example *yocto_relay*. After a little additional work, Yoctopuce is then able to generate all other files. Therefore, we highly recommend that you contact Yoctopuce support before undertaking to port the Yoctopuce library in another language. Collaborative work is advantageous to both parties.

19. High-level API Reference

This chapter summarizes the high-level API functions to drive your Yocto-4-20mA-Rx. 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.

19.1. General functions

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

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

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

Global functions

yCheckLogicalName(name)

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

yDisableExceptions()

Disables the use of exceptions to report runtime errors.

yEnableExceptions()

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

yEnableUSBHost(osContext)

This function is used only on Android.

yFreeAPI()

Frees dynamically allocated memory blocks used by the Yoctopuce library.

yGetAPIVersion()

Returns the version identifier for the Yoctopuce library in use.

yGetTickCount()

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

yHandleEvents(errmsg)

Maintains the device-to-library communication channel.

yInitAPI(mode, errmsg)

Initializes the Yoctopuce programming library explicitly.

yPreregisterHub(url, errmsg)

Fault-tolerant alternative to `RegisterHub()`.

yRegisterDeviceArrivalCallback(arrivalCallback)

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

yRegisterDeviceRemovalCallback(removalCallback)

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

yRegisterHub(url, errmsg)

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

yRegisterHubDiscoveryCallback(callback)

Register a callback function, to be called each time a network hub or a VirtualHub is detected on the local network.

yRegisterLogFunction(logfun)

Registers a log callback function.

ySelectArchitecture(arch)

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

ySetDelegate(object)

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

ySetTimeout(callback, ms_timeout, arguments)

Invoke the specified callback function after a given timeout.

ySleep(ms_duration, errmsg)

Pauses the execution flow for a specified duration.

yUnregisterHub(url)

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

yUpdateDeviceList(errmsg)

Triggers a (re)detection of connected Yoctopuce modules.

yUpdateDeviceList_async(callback, context)

Triggers a (re)detection of connected Yoctopuce modules.

YAPI.CheckLogicalName() yCheckLogicalName()

YAPI

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

js	function yCheckLogicalName (name)
nodejs	function CheckLogicalName (name)
php	function yCheckLogicalName (\$name)
cpp	bool yCheckLogicalName (const string& name)
m	BOOL yCheckLogicalName (NSString * name)
pas	function yCheckLogicalName (name : string): boolean
vb	function yCheckLogicalName (ByVal name As String) As Boolean
cs	bool CheckLogicalName (string name)
java	boolean CheckLogicalName (String name)
py	def CheckLogicalName (name)

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

Parameters :

name a string containing the name to check.

Returns :

true if the name is valid, false otherwise.

YAPI.DisableExceptions() yDisableExceptions()

YAPI

Disables the use of exceptions to report runtime errors.

js	function yDisableExceptions ()
nodejs	function DisableExceptions ()
php	function yDisableExceptions ()
cpp	void yDisableExceptions ()
m	void yDisableExceptions ()
pas	procedure yDisableExceptions ()
vb	procedure yDisableExceptions ()
cs	void DisableExceptions ()
py	def DisableExceptions ()

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

YAPI.EnableExceptions() yEnableExceptions()

YAPI

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

js	function yEnableExceptions ()
nodejs	function EnableExceptions ()
php	function yEnableExceptions ()
cpp	void yEnableExceptions ()
m	void yEnableExceptions ()
pas	procedure yEnableExceptions ()
vb	procedure yEnableExceptions ()
cs	void EnableExceptions ()
py	def EnableExceptions ()

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

YAPI.EnableUSBHost() yEnableUSBHost()

YAPI

This function is used only on Android.

```
java synchronized static 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 ()
nodejs	function FreeAPI ()
php	function yFreeAPI ()
cpp	void yFreeAPI ()
m	void yFreeAPI ()
pas	procedure yFreeAPI ()
vb	procedure yFreeAPI ()
cs	void FreeAPI ()
java	synchronized static void FreeAPI ()
py	def FreeAPI ()

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

YAPI.GetAPIVersion() yGetAPIVersion()

YAPI

Returns the version identifier for the Yoctopuce library in use.

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

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

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

Returns :

a character string describing the library version.

YAPI.GetTickCount() yGetTickCount()

YAPI

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

js	function yGetTickCount ()
nodejs	function GetTickCount ()
php	function yGetTickCount ()
cpp	u64 yGetTickCount ()
m	u64 yGetTickCount ()
pas	function yGetTickCount (): u64
vb	function yGetTickCount () As Long
cs	ulong GetTickCount ()
java	long GetTickCount ()
py	def GetTickCount ()

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

Returns :

a long integer corresponding to the millisecond counter.

YAPI.HandleEvents() yHandleEvents()

YAPI

Maintains the device-to-library communication channel.

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

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

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

Parameters :

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

Returns :

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

YAPI.InitAPI() yInitAPI()

YAPI

Initializes the Yoctopuce programming library explicitly.

js	function yInitAPI (mode , errmsg)
nodejs	function InitAPI (mode , errmsg)
php	function yInitAPI (\$mode , &\$errmsg)
cpp	YRETCODE yInitAPI (int mode , string& errmsg)
m	YRETCODE yInitAPI (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	synchronized static int InitAPI (int mode)
py	def InitAPI (mode , errmsg =None)

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

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

Parameters :

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

Returns :

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

YAPI.PreregisterHub() yPreregisterHub()

YAPI

Fault-tolerant alternative to RegisterHub().

js	function yPreregisterHub (url, errmsg)
nodejs	function PreregisterHub (url, errmsg)
php	function yPreregisterHub (\$url, &\$errmsg)
cpp	YRETCODE yPreregisterHub (const string& url, string& errmsg)
m	YRETCODE yPreregisterHub (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	synchronized static int PreregisterHub (String url)
py	def PreregisterHub (url, errmsg=None)

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

Parameters :

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

Returns :

YAPI_SUCCESS when the call succeeds.

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

YAPI.RegisterDeviceArrivalCallback() yRegisterDeviceArrivalCallback()

YAPI

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

js	function yRegisterDeviceArrivalCallback (arrivalCallback)
nodejs	function RegisterDeviceArrivalCallback (arrivalCallback)
php	function yRegisterDeviceArrivalCallback (\$arrivalCallback)
cpp	void yRegisterDeviceArrivalCallback (yDeviceUpdateCallback arrivalCallback)
m	void yRegisterDeviceArrivalCallback (yDeviceUpdateCallback arrivalCallback)
pas	procedure yRegisterDeviceArrivalCallback (arrivalCallback : yDeviceUpdateFunc)
vb	procedure yRegisterDeviceArrivalCallback (ByVal arrivalCallback As yDeviceUpdateFunc)
cs	void RegisterDeviceArrivalCallback (yDeviceUpdateFunc arrivalCallback)
java	synchronized static void RegisterDeviceArrivalCallback (DeviceArrivalCallback arrivalCallback)
py	def RegisterDeviceArrivalCallback (arrivalCallback)

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

Parameters :

arrivalCallback a procedure taking a YModule parameter, or null

YAPI.RegisterDeviceRemovalCallback() yRegisterDeviceRemovalCallback()

YAPI

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

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

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

Parameters :

removalCallback a procedure taking a YModule parameter, or null

YAPI.RegisterHub() yRegisterHub()

YAPI

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

```

js function yRegisterHub( url, errmsg)
nodejs function RegisterHub( url, errmsg)
php function yRegisterHub( $url, &$errmsg)
cpp YRETCODE yRegisterHub( const string& url, string& errmsg)
m YRETCODE yRegisterHub( 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 synchronized static int RegisterHub( String url)
py def RegisterHub( url, errmsg=None)

```

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

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

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

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

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

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

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

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

Parameters :

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

Returns :

YAPI_SUCCESS when the call succeeds.

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

YAPI.RegisterHubDiscoveryCallback() yRegisterHubDiscoveryCallback()

YAPI

Register a callback function, to be called each time a network hub or a VirtualHub is detected on the local network.

```
java void RegisterHubDiscoveryCallback( NewHubCallback callback)
```

Parameters :

callback a procedure taking a two string as parameter, or null

YAPI.RegisterLogFunction() yRegisterLogFunction()

YAPI

Registers a log callback function.

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

This callback will be called each time the API have something to say. Quite usefull 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.SetDelegate() ySetDelegate()

YAPI

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

```
m void ySetDelegate( id object)
```

The methods `yDeviceArrival` and `yDeviceRemoval` will be invoked while `yUpdateDeviceList` is running. You will have to call this function on a regular basis.

Parameters :

object an object that must follow the protocol YAPIDelegate, or nil

YAPI.SetTimeout() ySetTimeout()

YAPI

Invoke the specified callback function after a given timeout.

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

This function behaves more or less like Javascript `setTimeout`, but during the waiting time, it will call `yHandleEvents` and `yUpdateDeviceList` periodically, in order to keep the API up-to-date with current devices.

Parameters :

- callback** the function to call after the timeout occurs. On Microsoft Internet Explorer, the callback must be provided as a string to be evaluated.
- ms_timeout** an integer corresponding to the duration of the timeout, in milliseconds.
- arguments** additional arguments to be passed to the callback function can be provided, if needed (not supported on Microsoft Internet Explorer).

Returns :

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

YAPI.Sleep() ySleep()

Pauses the execution flow for a specified duration.

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

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

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

Parameters :

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

Returns :

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

YAPI.UnregisterHub() yUnregisterHub()

YAPI

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

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

Parameters :

url a string containing either "usb" or the

YAPI.UpdateDeviceList() yUpdateDeviceList()

YAPI

Triggers a (re)detection of connected Yoctopuce modules.

js	function yUpdateDeviceList (errmsg)
nodejs	function UpdateDeviceList (errmsg)
php	function yUpdateDeviceList (&\$errmsg)
cpp	YRETCODE yUpdateDeviceList (string& errmsg)
m	YRETCODE yUpdateDeviceList (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 ()
py	def UpdateDeviceList (errmsg =None)

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

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

Parameters :

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

Returns :

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

YAPI.UpdateDeviceList_async() yUpdateDeviceList_async()

YAPI

Triggers a (re)detection of connected Yoctopuce modules.

```
js function yUpdateDeviceList_async( callback, context)
nodejs function UpdateDeviceList_async( callback, context)
```

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

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

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox Javascript VM that does not implement context switching during blocking I/O calls.

Parameters :

- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the result code (`YAPI_SUCCESS` if the operation completes successfully) and the error message.
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

19.2. Module control interface

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

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

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

Global functions

yFindModule(funcn)

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

yFirstModule()

Starts the enumeration of modules currently accessible.

YModule methods

module→describe()

Returns a descriptive text that identifies the module.

module→download(pathname)

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

module→functionCount()

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

module→functionId(functionIndex)

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

module→functionName(functionIndex)

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

module→functionValue(functionIndex)

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

module→get_beacon()

Returns the state of the localization beacon.

module→get_errorMessage()

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

module→get_errorType()

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

module→get_firmwareRelease()

Returns the version of the firmware embedded in the module.

module→get_hardwareId()

Returns the unique hardware identifier of the module.

module→get_icon2d()

Returns the icon of the module.
module→get_lastLogs() Returns a string with last logs of the module.
module→get_logicalName() Returns the logical name of the module.
module→get_luminosity() Returns the luminosity of the module informative leds (from 0 to 100).
module→get_persistentSettings() Returns the current state of persistent module settings.
module→get_productId() Returns the USB device identifier of the module.
module→get_productName() Returns the commercial name of the module, as set by the factory.
module→get_productRelease() Returns the hardware release version of the module.
module→get_rebootCountdown() Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.
module→get_serialNumber() Returns the serial number of the module, as set by the factory.
module→get_upTime() Returns the number of milliseconds spent since the module was powered on.
module→get_usbBandwidth() Returns the number of USB interfaces used by the module.
module→get_usbCurrent() Returns the current consumed by the module on the USB bus, in milli-amps.
module→get_userData() Returns the value of the userData attribute, as previously stored using method <code>set_userData</code> .
module→isOnline() Checks if the module is currently reachable, without raising any error.
module→isOnline_async(callback, context) Checks if the module is currently reachable, without raising any error.
module→load(msValidity) Preloads the module cache with a specified validity duration.
module→load_async(msValidity, callback, context) Preloads the module cache with a specified validity duration (asynchronous version).
module→nextModule() Continues the module enumeration started using <code>yFirstModule()</code> .
module→reboot(secBeforeReboot) Schedules a simple module reboot after the given number of seconds.
module→revertFromFlash() Reloads the settings stored in the nonvolatile memory, as when the module is powered on.
module→saveToFlash() Saves current settings in the nonvolatile memory of the module.
module→set_beacon(newval) Turns on or off the module localization beacon.

module→set_logicalName(newval)

Changes the logical name of the module.

module→set_luminosity(newval)

Changes the luminosity of the module informative leds.

module→set_usbBandwidth(newval)

Changes the number of USB interfaces used by the module.

module→set_userData(data)

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

module→triggerFirmwareUpdate(secBeforeReboot)

Schedules a module reboot into special firmware update mode.

module→wait_async(callback, context)

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

YModule.FindModule() yFindModule()

YModule

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

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

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

Parameters :

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

Returns :

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

YModule.FirstModule() yFirstModule()

YModule

Starts the enumeration of modules currently accessible.

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

Use the method `YModule.nextModule()` to iterate on the next modules.

Returns :

a pointer to a YModule object, corresponding to the first module currently online, or a null pointer if there are none.

module→describe()**YModule**

Returns a descriptive text that identifies the module.

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

The text may include either the logical name or the serial number of the module.

Returns :

a string that describes the module

module→download()**YModule**

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

js	function download (pathname)
nodejs	function download (pathname)
php	function download (\$pathname)
cpp	string download (string pathname)
m	-(NSData*) download : (NSString*) pathname
pas	function download (pathname : string): TByteArray
vb	function download () As Byte
py	def download (pathname)
cmd	YModule target download pathname

Parameters :

pathname name of the new file to load

Returns :

a binary buffer with the file content

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

module→functionCount()**YModule**

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

js	function functionCount ()
nodejs	function functionCount ()
php	function functionCount ()
cpp	int functionCount ()
m	-(int) functionCount
pas	function functionCount (): integer
vb	function functionCount () As Integer
cs	int functionCount ()
py	def functionCount ()

Returns :

the number of functions on the module

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

module→**functionId()****YModule**

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

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

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a string corresponding to the unambiguous hardware identifier of the requested module function

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

module→**functionName()****YModule**

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

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

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a string corresponding to the logical name of the requested module function

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

module→**functionValue()****YModule**

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

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

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a short string (up to 6 characters) corresponding to the advertised value of the requested module function

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

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

Returns the state of the localization beacon.

js	function get_beacon ()
nodejs	function get_beacon ()
php	function get_beacon ()
cpp	Y_BEACON_enum get_beacon ()
m	-(Y_BEACON_enum) beacon
pas	function get_beacon (): Integer
vb	function get_beacon () As Integer
cs	int get_beacon ()
java	int get_beacon ()
py	def get_beacon ()
cmd	YModule target get_beacon

Returns :

either Y_BEACON_OFF or Y_BEACON_ON, according to the state of the localization beacon

On failure, throws an exception or returns Y_BEACON_INVALID.

module→**get_errorMessage()****YModule****module**→**errorMessage()**

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

<code>js</code>	<code>function get_errorMessage()</code>
<code>nodejs</code>	<code>function get_errorMessage()</code>
<code>php</code>	<code>function get_errorMessage()</code>
<code>cpp</code>	<code>string get_errorMessage()</code>
<code>m</code>	<code>-(NSString*) errorMessage</code>
<code>pas</code>	<code>function get_errorMessage(): string</code>
<code>vb</code>	<code>function get_errorMessage() As String</code>
<code>cs</code>	<code>string get_errorMessage()</code>
<code>java</code>	<code>String get_errorMessage()</code>
<code>py</code>	<code>def get_errorMessage()</code>

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

Returns :

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

module→**get_errorType()**
module→**errorType()**

YModule

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

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

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

Returns :

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

module→get_firmwareRelease() module→firmwareRelease()

YModule

Returns the version of the firmware embedded in the module.

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

Returns :

a string corresponding to the version of the firmware embedded in the module

On failure, throws an exception or returns Y_FIRMWARERELEASE_INVALID.

module→**get_hardwareId()****YModule****module**→**hardwareId()**

Returns the unique hardware identifier of the module.

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

The unique hardware identifier is 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 ()
nodejs	function get_icon2d ()
php	function get_icon2d ()
cpp	string get_icon2d ()
m	-(NSData*) icon2d
pas	function get_icon2d (): TByteArray
vb	function get_icon2d () As Byte
py	def get_icon2d ()
cmd	YModule target get_icon2d

The icon is a PNG image and does not exceeds 1536 bytes.

Returns :

a binary buffer with module icon, in png format.

module→get_lastLogs()**YModule****module→lastLogs()**

Returns a string with last logs of the module.

js	function get_lastLogs ()
nodejs	function get_lastLogs ()
php	function get_lastLogs ()
cpp	string get_lastLogs ()
m	-(NSString*) lastLogs
pas	function get_lastLogs (): string
vb	function get_lastLogs () As String
cs	string get_lastLogs ()
java	String get_lastLogs ()
py	def get_lastLogs ()
cmd	YModule target get_lastLogs

This method return only logs that are still in the module.

Returns :

a string with last logs of the module.

module→**get_logicalName()****YModule****module**→**logicalName()**

Returns the logical name of the module.

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

Returns :

a string corresponding to the logical name of the module

On failure, throws an exception or returns Y_LOGICALNAME_INVALID.

module→get_luminosity()
module→luminosity()

YModule

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

js	function get_luminosity ()
nodejs	function get_luminosity ()
php	function get_luminosity ()
cpp	int get_luminosity ()
m	-(int) luminosity
pas	function get_luminosity (): LongInt
vb	function get_luminosity () As Integer
cs	int get_luminosity ()
java	int get_luminosity ()
py	def get_luminosity ()
cmd	YModule target get_luminosity

Returns :

an integer corresponding to the luminosity of the module informative leds (from 0 to 100)

On failure, throws an exception or returns Y_LUMINOSITY_INVALID.

module→**get_persistentSettings()****YModule****module**→**persistentSettings()**

Returns the current state of persistent module settings.

js	function get_persistentSettings ()
nodejs	function get_persistentSettings ()
php	function get_persistentSettings ()
cpp	Y_PERSISTENTSETTINGS_enum get_persistentSettings ()
m	-(Y_PERSISTENTSETTINGS_enum) persistentSettings
pas	function get_persistentSettings (): Integer
vb	function get_persistentSettings () As Integer
cs	int get_persistentSettings ()
java	int get_persistentSettings ()
py	def get_persistentSettings ()
cmd	YModule target get_persistentSettings

Returns :

a value among Y_PERSISTENTSETTINGS_LOADED, Y_PERSISTENTSETTINGS_SAVED and Y_PERSISTENTSETTINGS_MODIFIED corresponding to the current state of persistent module settings

On failure, throws an exception or returns Y_PERSISTENTSETTINGS_INVALID.

module→**get_productId()****YModule****module**→**productId()**

Returns the USB device identifier of the module.

js	function get_productId ()
nodejs	function get_productId ()
php	function get_productId ()
cpp	int get_productId ()
m	-(int) productId
pas	function get_productId (): LongInt
vb	function get_productId () As Integer
cs	int get_productId ()
java	int get_productId ()
py	def get_productId ()
cmd	YModule target get_productId

Returns :

an integer corresponding to the USB device identifier of the module

On failure, throws an exception or returns Y_PRODUCTID_INVALID.

module→**get_productName()****YModule****module**→**productName()**

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

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

Returns :

a string corresponding to the commercial name of the module, as set by the factory

On failure, throws an exception or returns Y_PRODUCTNAME_INVALID.

module→get_productRelease()**YModule****module→productRelease()**

Returns the hardware release version of the module.

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

Returns :

an integer corresponding to the hardware release version of the module

On failure, throws an exception or returns Y_PRODUCTRELEASE_INVALID.

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

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

js	function get_rebootCountdown ()
nodejs	function get_rebootCountdown ()
php	function get_rebootCountdown ()
cpp	int get_rebootCountdown ()
m	-(int) rebootCountdown
pas	function get_rebootCountdown (): LongInt
vb	function get_rebootCountdown () As Integer
cs	int get_rebootCountdown ()
java	int get_rebootCountdown ()
py	def get_rebootCountdown ()
cmd	YModule target get_rebootCountdown

Returns :

an integer corresponding to the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled

On failure, throws an exception or returns Y_REBOOTCOUNTDOWN_INVALID.

module→**get_serialNumber()****YModule****module**→**serialNumber()**

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

js	function get_serialNumber ()
nodejs	function get_serialNumber ()
php	function get_serialNumber ()
cpp	string get_serialNumber ()
m	-(NSString*) serialNumber
pas	function get_serialNumber (): string
vb	function get_serialNumber () As String
cs	string get_serialNumber ()
java	String get_serialNumber ()
py	def get_serialNumber ()
cmd	YModule target get_serialNumber

Returns :

a string corresponding to the serial number of the module, as set by the factory

On failure, throws an exception or returns Y_SERIALNUMBER_INVALID.

module→**get_upTime()****YModule****module**→**upTime()**

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

js	function get_upTime ()
nodejs	function get_upTime ()
php	function get_upTime ()
cpp	s64 get_upTime ()
m	-(s64) upTime
pas	function get_upTime (): int64
vb	function get_upTime () As Long
cs	long get_upTime ()
java	long get_upTime ()
py	def get_upTime ()
cmd	YModule target get_upTime

Returns :

an integer corresponding to the number of milliseconds spent since the module was powered on

On failure, throws an exception or returns Y_UPTIME_INVALID.

module→get_usbBandwidth()**YModule****module→usbBandwidth()**

Returns the number of USB interfaces used by the module.

js	function get_usbBandwidth ()
nodejs	function get_usbBandwidth ()
php	function get_usbBandwidth ()
cpp	Y_USBBANDWIDTH_enum get_usbBandwidth ()
m	-(Y_USBBANDWIDTH_enum) usbBandwidth
pas	function get_usbBandwidth (): Integer
vb	function get_usbBandwidth () As Integer
cs	int get_usbBandwidth ()
java	int get_usbBandwidth ()
py	def get_usbBandwidth ()
cmd	YModule target get_usbBandwidth

Returns :

either Y_USBBANDWIDTH_SIMPLE or Y_USBBANDWIDTH_DOUBLE, according to the number of USB interfaces used by the module

On failure, throws an exception or returns Y_USBBANDWIDTH_INVALID.

module→**get_usbCurrent()****YModule****module**→**usbCurrent()**

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

js	function get_usbCurrent ()
nodejs	function get_usbCurrent ()
php	function get_usbCurrent ()
cpp	int get_usbCurrent ()
m	-(int) usbCurrent
pas	function get_usbCurrent (): LongInt
vb	function get_usbCurrent () As Integer
cs	int get_usbCurrent ()
java	int get_usbCurrent ()
py	def get_usbCurrent ()
cmd	YModule target get_usbCurrent

Returns :

an integer corresponding to the current consumed by the module on the USB bus, in milli-amps

On failure, throws an exception or returns Y_USBCURRENT_INVALID.

module→**get_userData()****YModule****module**→**userData()**

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

js	function get_userData ()
nodejs	function get_userData ()
php	function get_userData ()
cpp	void * get_userData ()
m	-(void*) userData
pas	function get_userData (): Tobject
vb	function get_userData () As Object
cs	object get_userData ()
java	Object get_userData ()
py	def get_userData ()

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

Returns :

the object stored previously by the caller.

module→isOnline()**YModule**

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

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

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

Returns :

true if the module can be reached, and false otherwise

module→isOnline_async()**YModule**

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

```
js function isOnline_async( callback, context)
nodejs function isOnline_async( callback, context)
```

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox Javascript VM that does not implement context switching during blocking I/O calls.

Parameters :

- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving module object and the boolean result
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

module→load()**YModule**

Preloads the module cache with a specified validity duration.

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

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

Parameters :

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

Returns :

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

module→load_async()**YModule**

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

```
js function load_async( msValidity, callback, context)
nodejs function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all module attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

- msValidity** an integer corresponding to the validity of the loaded module parameters, in milliseconds
- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving module object and the error code (or YAPI_SUCCESS)
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

module→nextModule()**YModule**

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

<code>js</code>	<code>function nextModule()</code>
<code>nodejs</code>	<code>function nextModule()</code>
<code>php</code>	<code>function nextModule()</code>
<code>cpp</code>	<code>YModule * nextModule()</code>
<code>m</code>	<code>-(YModule*) nextModule</code>
<code>pas</code>	<code>function nextModule(): TYModule</code>
<code>vb</code>	<code>function nextModule() As YModule</code>
<code>cs</code>	<code>YModule nextModule()</code>
<code>java</code>	<code>YModule nextModule()</code>
<code>py</code>	<code>def nextModule()</code>

Returns :

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

module→reboot()**YModule**

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

js	function reboot (secBeforeReboot)
nodejs	function reboot (secBeforeReboot)
php	function reboot (\$secBeforeReboot)
cpp	int reboot (int secBeforeReboot)
m	-(int) reboot : (int) secBeforeReboot
pas	function reboot (secBeforeReboot : LongInt): LongInt
vb	function reboot () As Integer
cs	int reboot (int secBeforeReboot)
java	int reboot (int secBeforeReboot)
py	def reboot (secBeforeReboot)
cmd	YModule target reboot secBeforeReboot

Parameters :

secBeforeReboot number of seconds before rebooting

Returns :

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

module→revertFromFlash()**YModule**

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

<code>js</code>	<code>function revertFromFlash()</code>
<code>nodejs</code>	<code>function revertFromFlash()</code>
<code>php</code>	<code>function revertFromFlash()</code>
<code>cpp</code>	<code>int revertFromFlash()</code>
<code>m</code>	<code>-(int) revertFromFlash</code>
<code>pas</code>	<code>function revertFromFlash(): LongInt</code>
<code>vb</code>	<code>function revertFromFlash() As Integer</code>
<code>cs</code>	<code>int revertFromFlash()</code>
<code>java</code>	<code>int revertFromFlash()</code>
<code>py</code>	<code>def revertFromFlash()</code>
<code>cmd</code>	<code>YModule target revertFromFlash</code>

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 ()
nodejs	function saveToFlash ()
php	function saveToFlash ()
cpp	int saveToFlash ()
m	-(int) saveToFlash
pas	function saveToFlash (): LongInt
vb	function saveToFlash () As Integer
cs	int saveToFlash ()
java	int saveToFlash ()
py	def saveToFlash ()
cmd	YModule target saveToFlash

Warning: the number of allowed save operations during a module life is limited (about 100000 cycles). Do not call this function within a loop.

Returns :

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

module→**set_beacon()****YModule****module**→**setBeacon()**

Turns on or off the module localization beacon.

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

Parameters :

newval either Y_BEACON_OFF or Y_BEACON_ON

Returns :

YAPI_SUCCESS if the call succeeds.

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

module→**set_logicalName()****YModule****module**→**setLogicalName()**

Changes the logical name of the module.

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

The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer corresponding to the luminosity of the module informative leds

Returns :

YAPI_SUCCESS if the call succeeds.

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

module→**set_usbBandwidth()****YModule****module**→**setUsbBandwidth()**

Changes the number of USB interfaces used by the module.

js	function set_usbBandwidth (newval)
nodejs	function set_usbBandwidth (newval)
php	function set_usbBandwidth (\$newval)
cpp	int set_usbBandwidth (Y_USBBANDWIDTH_enum newval)
m	-(int) setUsbBandwidth : (Y_USBBANDWIDTH_enum) newval
pas	function set_usbBandwidth (newval : Integer): integer
vb	function set_usbBandwidth (ByVal newval As Integer) As Integer
cs	int set_usbBandwidth (int newval)
java	int set_usbBandwidth (int newval)
py	def set_usbBandwidth (newval)
cmd	YModule target set_usbBandwidth newval

You must reboot the module after changing this setting.

Parameters :

newval either Y_USBBANDWIDTH_SIMPLE or Y_USBBANDWIDTH_DOUBLE, according to the number of USB interfaces used by the module

Returns :

YAPI_SUCCESS if the call succeeds.

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

module→**set_userData()****YModule****module**→**setUserData()**

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

<code>js</code>	<code>function set_userData(data)</code>
<code>nodejs</code>	<code>function set_userData(data)</code>
<code>php</code>	<code>function set_userData(\$data)</code>
<code>cpp</code>	<code>void set_userData(void* data)</code>
<code>m</code>	<code>-(void) setUserData : (void*) data</code>
<code>pas</code>	<code>procedure set_userData(data: Tobject)</code>
<code>vb</code>	<code>procedure set_userData(ByVal data As Object)</code>
<code>cs</code>	<code>void set_userData(object data)</code>
<code>java</code>	<code>void set_userData(Object data)</code>
<code>py</code>	<code>def set_userData(data)</code>

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

Parameters :

data any kind of object to be stored

module→triggerFirmwareUpdate()**YModule**

Schedules a module reboot into special firmware update mode.

js	function triggerFirmwareUpdate (secBeforeReboot)
nodejs	function triggerFirmwareUpdate (secBeforeReboot)
php	function triggerFirmwareUpdate (\$secBeforeReboot)
cpp	int triggerFirmwareUpdate (int secBeforeReboot)
m	-(int) triggerFirmwareUpdate : (int) secBeforeReboot
pas	function triggerFirmwareUpdate (secBeforeReboot : LongInt): LongInt
vb	function triggerFirmwareUpdate () As Integer
cs	int triggerFirmwareUpdate (int secBeforeReboot)
java	int triggerFirmwareUpdate (int secBeforeReboot)
py	def triggerFirmwareUpdate (secBeforeReboot)
cmd	YModule target triggerFirmwareUpdate secBeforeReboot

Parameters :

secBeforeReboot number of seconds before rebooting

Returns :

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

module→**wait_async()****YModule**

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

```
js function wait_async( callback, context)
```

```
nodejs function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

19.3. GenericSensor function interface

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

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

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

Global functions

yFindGenericSensor(func)

Retrieves a generic sensor for a given identifier.

yFirstGenericSensor()

Starts the enumeration of generic sensors currently accessible.

YGenericSensor methods

genericsensor→calibrateFromPoints(rawValues, refValues)

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

genericsensor→describe()

Returns a short text that describes the generic sensor in the form
TYPE (NAME) = SERIAL . FUNCTIONID.

genericsensor→get_advertisedValue()

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

genericsensor→get_currentRawValue()

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

genericsensor→get_currentValue()

Returns the current measured value.

genericsensor→get_errorMessage()

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

genericsensor→get_errorType()

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

genericsensor→get_friendlyName()

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

genericsensor→get_functionDescriptor()

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

genericsensor→get_functionId()

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

genericsensor→get_hardwareId()

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

genericSensor→get_highestValue()

Returns the maximal value observed.

genericSensor→get_logFrequency()

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

genericSensor→get_logicalName()

Returns the logical name of the generic sensor.

genericSensor→get_lowestValue()

Returns the minimal value observed.

genericSensor→get_module()

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

genericSensor→get_module_async(callback, context)

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

genericSensor→get_recordedData(startTime, endTime)

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

genericSensor→get_reportFrequency()

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

genericSensor→get_resolution()

Returns the resolution of the measured values.

genericSensor→get_signalRange()

Returns the electric signal range used by the sensor.

genericSensor→get_signalUnit()

Returns the measuring unit of the electrical signal used by the sensor.

genericSensor→get_signalValue()

Returns the measured value of the electrical signal used by the sensor.

genericSensor→get_unit()

Returns the measuring unit for the measured value.

genericSensor→get_userData()

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

genericSensor→get_valueRange()

Returns the physical value range measured by the sensor.

genericSensor→isOnline()

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

genericSensor→isOnline_async(callback, context)

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

genericSensor→load(msValidity)

Preloads the generic sensor cache with a specified validity duration.

genericSensor→loadCalibrationPoints(rawValues, refValues)

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

genericSensor→load_async(msValidity, callback, context)

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

genericSensor→nextGenericSensor()

Continues the enumeration of generic sensors started using yFirstGenericSensor().

genericSensor→registerTimedReportCallback(callback)

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

genericsensor→registerValueCallback(callback)

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

genericsensor→set_highestValue(newval)

Changes the recorded maximal value observed.

genericsensor→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

genericsensor→set_logicalName(newval)

Changes the logical name of the generic sensor.

genericsensor→set_lowestValue(newval)

Changes the recorded minimal value observed.

genericsensor→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

genericsensor→set_resolution(newval)

Changes the resolution of the measured physical values.

genericsensor→set_signalRange(newval)

Changes the electric signal range used by the sensor.

genericsensor→set_unit(newval)

Changes the measuring unit for the measured value.

genericsensor→set_userData(data)

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

genericsensor→set_valueRange(newval)

Changes the physical value range measured by the sensor.

genericsensor→wait_async(callback, context)

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

YGenericSensor.FindGenericSensor() yFindGenericSensor()

YGenericSensor

Retrieves a generic sensor for a given identifier.

js	function yFindGenericSensor (func)
nodejs	function FindGenericSensor (func)
php	function yFindGenericSensor (\$func)
cpp	YGenericSensor* yFindGenericSensor (const string& func)
m	YGenericSensor* yFindGenericSensor (NSString* func)
pas	function yFindGenericSensor (func : string): TYGenericSensor
vb	function yFindGenericSensor (ByVal func As String) As YGenericSensor
cs	YGenericSensor FindGenericSensor (string func)
java	YGenericSensor FindGenericSensor (String func)
py	def FindGenericSensor (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 generic sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YGenericSensor.isOnline()` to test if the generic sensor is indeed online at a given time. In case of ambiguity when looking for a generic sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

func a string that uniquely characterizes the generic sensor

Returns :

a `YGenericSensor` object allowing you to drive the generic sensor.

YGenericSensor.FirstGenericSensor() yFirstGenericSensor()

YGenericSensor

Starts the enumeration of generic sensors currently accessible.

js	function yFirstGenericSensor ()
nodejs	function FirstGenericSensor ()
php	function yFirstGenericSensor ()
cpp	YGenericSensor* yFirstGenericSensor ()
m	YGenericSensor* yFirstGenericSensor ()
pas	function yFirstGenericSensor (): TYGenericSensor
vb	function yFirstGenericSensor () As YGenericSensor
cs	YGenericSensor FirstGenericSensor ()
java	YGenericSensor FirstGenericSensor ()
py	def FirstGenericSensor ()

Use the method `YGenericSensor.nextGenericSensor()` to iterate on next generic sensors.

Returns :

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

genericsensor→calibrateFromPoints()**YGenericSensor**

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

```

js function calibrateFromPoints( rawValues, refValues)
nodejs function calibrateFromPoints( rawValues, refValues)
php function calibrateFromPoints( $rawValues, $refValues)
cpp int calibrateFromPoints( vector<double> rawValues,
                             vector<double> refValues)
m -(int) calibrateFromPoints : (NSMutableArray*) rawValues
      : (NSMutableArray*) refValues
pas function calibrateFromPoints( rawValues: TDoubleArray,
                                 refValues: TDoubleArray): LongInt
vb procedure calibrateFromPoints( )
cs int calibrateFromPoints( List<double> rawValues,
                             List<double> refValues)
java int calibrateFromPoints( ArrayList<Double> rawValues,
                              ArrayList<Double> refValues)
py def calibrateFromPoints( rawValues, refValues)
cmd YGenericSensor target calibrateFromPoints rawValues refValues

```

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

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

Parameters :

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

genericsensor→describe()**YGenericSensor**

Returns a short text that describes the generic sensor in the form
`TYPE (NAME) = SERIAL . FUNCTIONID`.

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

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

Returns :

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

genericsensor→**get_advertisedValue()** **genericsensor**→**advertisedValue()**

YGenericSensor

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

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

Returns :

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

genericsensor→get_currentRawValue()
genericsensor→currentRawValue()

YGenericSensor

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

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

Returns :

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

On failure, throws an exception or returns Y_CURRENTRAWVALUE_INVALID.

genericSensor→**get_currentValue()** **genericSensor**→**currentValue()**

YGenericSensor

Returns the current measured value.

js	function get_currentValue ()
nodejs	function get_currentValue ()
php	function get_currentValue ()
cpp	double get_currentValue ()
m	-(double) currentValue
pas	function get_currentValue (): double
vb	function get_currentValue () As Double
cs	double get_currentValue ()
java	double get_currentValue ()
py	def get_currentValue ()
cmd	YGenericSensor target get_currentValue

Returns :

a floating point number corresponding to the current measured value

On failure, throws an exception or returns Y_CURRENTVALUE_INVALID.

genericsensor→get_errorMessage() **genericsensor→errorMessage()**

YGenericSensor

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

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

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

Returns :

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

genericsensor→get_errorType()
genericsensor→errorType()**YGenericSensor**

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

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

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

Returns :

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

genericsensor→get_friendlyName()
genericsensor→friendlyName()

YGenericSensor

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

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

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

Returns :

a string that uniquely identifies the generic sensor using logical names (ex: `MyCustomName.relay1`)
 On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

genericsensor→**get_functionDescriptor()** **genericsensor**→**functionDescriptor()**

YGenericSensor

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

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

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

Returns :

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

genericsensor→get_functionId() **genericsensor→functionId()**

YGenericSensor

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

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

For example `relay1`

Returns :

a string that identifies the generic sensor (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

genericsensor→**get_hardwareId()** **genericsensor**→**hardwareId()**

YGenericSensor

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

js	function get_hardwareId ()
nodejs	function get_hardwareId ()
php	function get_hardwareId ()
c++	string get_hardwareId ()
m	-(NSString*) hardwareId
vb	function get_hardwareId () As String
cs	string get_hardwareId ()
java	String get_hardwareId ()
py	def get_hardwareId ()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the generic sensor. (for example `RELAYLO1-123456.relay1`)

Returns :

a string that uniquely identifies the generic sensor (ex: `RELAYLO1-123456.relay1`) On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

genericsensor→get_highestValue()
genericsensor→highestValue()

YGenericSensor

Returns the maximal value observed.

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

Returns :

a floating point number corresponding to the maximal value observed

On failure, throws an exception or returns Y_HIGHESTVALUE_INVALID.

genericsensor→get_logFrequency() **genericsensor→logFrequency()**

YGenericSensor

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

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

Returns :

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

On failure, throws an exception or returns Y_LOGFREQUENCY_INVALID.

genericsensor→get_logicalName() **genericsensor→logicalName()**

YGenericSensor

Returns the logical name of the generic sensor.

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

Returns :

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

genericsensor→**get_lowestValue()** **genericsensor**→**lowestValue()**

YGenericSensor

Returns the minimal value observed.

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

Returns :

a floating point number corresponding to the minimal value observed

On failure, throws an exception or returns Y_LOWESTVALUE_INVALID.

genericsensor→get_module() **genericsensor→module()**

YGenericSensor

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

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

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

Returns :

an instance of YModule

genericsensor→get_module_async()
genericsensor→module_async()**YGenericSensor**

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

js	function get_module_async (callback , context)
nodejs	function get_module_async (callback , context)

If the function cannot be located on any module, the returned YModule object does not show as 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.

genericsensor→get_recordedData() genericsensor→recordedData()

YGenericSensor

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

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

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

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

Parameters :

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

Returns :

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

genericsensor→get_reportFrequency() **genericsensor→reportFrequency()**

YGenericSensor

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

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

Returns :

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

On failure, throws an exception or returns Y_REPORTFREQUENCY_INVALID.

genericsensor→**get_resolution()**
genericsensor→**resolution()**

YGenericSensor

Returns the resolution of the measured values.

js	function get_resolution ()
nodejs	function get_resolution ()
php	function get_resolution ()
cpp	double get_resolution ()
m	-(double) resolution
pas	function get_resolution (): double
vb	function get_resolution () As Double
cs	double get_resolution ()
java	double get_resolution ()
py	def get_resolution ()
cmd	YGenericSensor target get_resolution

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

Returns :

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

On failure, throws an exception or returns Y_RESOLUTION_INVALID.

genericsensor→get_signalRange()
genericsensor→signalRange()**YGenericSensor**

Returns the electric signal range used by the sensor.

js	function get_signalRange ()
nodejs	function get_signalRange ()
php	function get_signalRange ()
cpp	string get_signalRange ()
m	-(NSString*) signalRange
pas	function get_signalRange (): string
vb	function get_signalRange () As String
cs	string get_signalRange ()
java	String get_signalRange ()
py	def get_signalRange ()
cmd	YGenericSensor target get_signalRange

Returns :

a string corresponding to the electric signal range used by the sensor

On failure, throws an exception or returns Y_SIGNALRANGE_INVALID.

genericsensor→get_signalUnit()
genericsensor→signalUnit()

YGenericSensor

Returns the measuring unit of the electrical signal used by the sensor.

js	function get_signalUnit ()
nodejs	function get_signalUnit ()
php	function get_signalUnit ()
cpp	string get_signalUnit ()
m	-(NSString*) signalUnit
pas	function get_signalUnit (): string
vb	function get_signalUnit () As String
cs	string get_signalUnit ()
java	String get_signalUnit ()
py	def get_signalUnit ()
cmd	YGenericSensor target get_signalUnit

Returns :

a string corresponding to the measuring unit of the electrical signal used by the sensor

On failure, throws an exception or returns Y_SIGNALUNIT_INVALID.

genericSensor→get_signalValue()
genericSensor→signalValue()**YGenericSensor**

Returns the measured value of the electrical signal used by the sensor.

js	function get_signalValue ()
nodejs	function get_signalValue ()
php	function get_signalValue ()
cpp	double get_signalValue ()
m	-(double) signalValue
pas	function get_signalValue (): double
vb	function get_signalValue () As Double
cs	double get_signalValue ()
java	double get_signalValue ()
py	def get_signalValue ()
cmd	YGenericSensor target get_signalValue

Returns :

a floating point number corresponding to the measured value of the electrical signal used by the sensor

On failure, throws an exception or returns Y_SIGNALVALUE_INVALID.

genericsensor→**get_unit()**
genericsensor→**unit()**

YGenericSensor

Returns the measuring unit for the measured value.

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

Returns :

a string corresponding to the measuring unit for the measured value

On failure, throws an exception or returns Y_UNIT_INVALID.

genericsensor→get_userdata()
genericsensor→userData()**YGenericSensor**

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

js	function get_userdata ()
nodejs	function get_userdata ()
php	function get_userdata ()
cpp	void * get_userdata ()
m	-(void*) userData
pas	function get_userdata (): Tobject
vb	function get_userdata () As Object
cs	object get_userdata ()
java	Object get_userdata ()
py	def get_userdata ()

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

Returns :

the object stored previously by the caller.

genericsensor→get_valueRange() **genericsensor→valueRange()**

YGenericSensor

Returns the physical value range measured by the sensor.

js	function get_valueRange ()
nodejs	function get_valueRange ()
php	function get_valueRange ()
cpp	string get_valueRange ()
m	-(NSString*) valueRange
pas	function get_valueRange (): string
vb	function get_valueRange () As String
cs	string get_valueRange ()
java	String get_valueRange ()
py	def get_valueRange ()
cmd	YGenericSensor target get_valueRange

Returns :

a string corresponding to the physical value range measured by the sensor

On failure, throws an exception or returns Y_VALUERANGE_INVALID.

genericsensor→isOnline()**YGenericSensor**

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

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

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

Returns :

true if the generic sensor can be reached, and false otherwise

genericsensor→isOnline_async()**YGenericSensor**

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

```
js function isOnline_async( callback, context)
```

```
nodejs function isOnline_async( callback, context)
```

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

genericsensor→load()**YGenericSensor**

Preloads the generic sensor cache with a specified validity duration.

js	function load (msValidity)
nodejs	function load (msValidity)
php	function load (\$msValidity)
cpp	YRETCODE load (int msValidity)
m	-(YRETCODE) load : (int) msValidity
pas	function load (msValidity : integer): YRETCODE
vb	function load (ByVal msValidity As Integer) As YRETCODE
cs	YRETCODE load (int msValidity)
java	int load (long msValidity)
py	def load (msValidity)

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

Parameters :

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

Returns :

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

genericsensor→loadCalibrationPoints()**YGenericSensor**

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

```

js    function loadCalibrationPoints( rawValues, refValues)
nodejs function loadCalibrationPoints( rawValues, refValues)
php    function loadCalibrationPoints( &$rawValues, &$refValues)
cpp    int loadCalibrationPoints( vector<double>& rawValues,
                                vector<double>& refValues)

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

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

vb      procedure loadCalibrationPoints( )
cs      int loadCalibrationPoints( List<double> rawValues,
                                List<double> refValues)

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

py      def loadCalibrationPoints( rawValues, refValues)
cmd      YGenericSensor target loadCalibrationPoints rawValues refValues

```

Parameters :

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

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

genericsensor→load_async()**YGenericSensor**

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

```
js function load_async( msValidity, callback, context)
nodejs function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

- msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds
- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or YAPI_SUCCESS)
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

genericSensor→**nextGenericSensor()****YGenericSensor**

Continues the enumeration of generic sensors started using `yFirstGenericSensor()`.

<code>js</code>	<code>function nextGenericSensor()</code>
<code>nodejs</code>	<code>function nextGenericSensor()</code>
<code>php</code>	<code>function nextGenericSensor()</code>
<code>cpp</code>	<code>YGenericSensor * nextGenericSensor()</code>
<code>m</code>	<code>-(YGenericSensor*) nextGenericSensor</code>
<code>pas</code>	<code>function nextGenericSensor(): TYGenericSensor</code>
<code>vb</code>	<code>function nextGenericSensor() As YGenericSensor</code>
<code>cs</code>	<code>YGenericSensor nextGenericSensor()</code>
<code>java</code>	<code>YGenericSensor nextGenericSensor()</code>
<code>py</code>	<code>def nextGenericSensor()</code>

Returns :

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

genericsensor→registerTimedReportCallback()**YGenericSensor**

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

js	function registerTimedReportCallback (callback)
nodejs	function registerTimedReportCallback (callback)
php	function registerTimedReportCallback (\$callback)
cpp	int registerTimedReportCallback (YGenericSensorTimedReportCallback callback)
m	-(int) registerTimedReportCallback : (YGenericSensorTimedReportCallback) callback
pas	function registerTimedReportCallback (callback : TYGenericSensorTimedReportCallback): LongInt
vb	function registerTimedReportCallback () As Integer
cs	int registerTimedReportCallback (TimedReportCallback callback)
java	int registerTimedReportCallback (TimedReportCallback callback)
py	def registerTimedReportCallback (callback)

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

Parameters :

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

genericsensor→registerValueCallback()**YGenericSensor**

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

js	function registerValueCallback (callback)
nodejs	function registerValueCallback (callback)
php	function registerValueCallback (\$callback)
cpp	int registerValueCallback (YGenericSensorValueCallback callback)
m	-(int) registerValueCallback : (YGenericSensorValueCallback) callback
pas	function registerValueCallback (callback : TYGenericSensorValueCallback): LongInt
vb	function registerValueCallback () As Integer
cs	int registerValueCallback (ValueCallback callback)
java	int registerValueCallback (UpdateCallback callback)
py	def registerValueCallback (callback)

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

Parameters :

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

genericSensor→set_highestValue() **genericSensor→setHighestValue()**

YGenericSensor

Changes the recorded maximal value observed.

js	function set_highestValue (newval)
nodejs	function set_highestValue (newval)
php	function set_highestValue (\$newval)
cpp	int set_highestValue (double newval)
m	-(int) setHighestValue : (double) newval
pas	function set_highestValue (newval : double): integer
vb	function set_highestValue (ByVal newval As Double) As Integer
cs	int set_highestValue (double newval)
java	int set_highestValue (double newval)
py	def set_highestValue (newval)
cmd	YGenericSensor target set_highestValue newval

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

genericsensor→set_logFrequency() **genericsensor→setLogFrequency()**

YGenericSensor

Changes the datalogger recording frequency for this function.

js	function set_logFrequency (newval)
nodejs	function set_logFrequency (newval)
php	function set_logFrequency (\$newval)
cpp	int set_logFrequency (const string& newval)
m	-(int) setLogFrequency : (NSString*) newval
pas	function set_logFrequency (newval : string): integer
vb	function set_logFrequency (ByVal newval As String) As Integer
cs	int set_logFrequency (string newval)
java	int set_logFrequency (String newval)
py	def set_logFrequency (newval)
cmd	YGenericSensor target set_logFrequency newval

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

genericsensor→**set_logicalName()** **genericsensor**→**setLogicalName()**

YGenericSensor

Changes the logical name of the generic sensor.

js	function set_logicalName (newval)
nodejs	function set_logicalName (newval)
php	function set_logicalName (\$newval)
cpp	int set_logicalName (const string& newval)
m	-(int) setLogicalName : (NSString*) newval
pas	function set_logicalName (newval : string): integer
vb	function set_logicalName (ByVal newval As String) As Integer
cs	int set_logicalName (string newval)
java	int set_logicalName (String newval)
py	def set_logicalName (newval)
cmd	YGenericSensor target set_logicalName newval

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

Parameters :

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

Returns :

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

genericsensor→set_lowestValue() **genericsensor→setLowestValue()**

YGenericSensor

Changes the recorded minimal value observed.

js	function set_lowestValue (newval)
nodejs	function set_lowestValue (newval)
php	function set_lowestValue (\$newval)
cpp	int set_lowestValue (double newval)
m	-(int) setLowestValue : (double) newval
pas	function set_lowestValue (newval : double): integer
vb	function set_lowestValue (ByVal newval As Double) As Integer
cs	int set_lowestValue (double newval)
java	int set_lowestValue (double newval)
py	def set_lowestValue (newval)
cmd	YGenericSensor target set_lowestValue newval

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

genericsensor→set_reportFrequency() **genericsensor→setReportFrequency()**

YGenericSensor

Changes the timed value notification frequency for this function.

js	function set_reportFrequency (newval)
nodejs	function set_reportFrequency (newval)
php	function set_reportFrequency (\$newval)
cpp	int set_reportFrequency (const string& newval)
m	-(int) setReportFrequency : (NSString*) newval
pas	function set_reportFrequency (newval : string): integer
vb	function set_reportFrequency (ByVal newval As String) As Integer
cs	int set_reportFrequency (string newval)
java	int set_reportFrequency (String newval)
py	def set_reportFrequency (newval)
cmd	YGenericSensor target set_reportFrequency newval

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

genericsensor→set_resolution() **genericsensor→setResolution()**

YGenericSensor

Changes the resolution of the measured physical values.

js	function set_resolution (newval)
nodejs	function set_resolution (newval)
php	function set_resolution (\$newval)
cpp	int set_resolution (double newval)
m	-(int) setResolution : (double) newval
pas	function set_resolution (newval : double): integer
vb	function set_resolution (ByVal newval As Double) As Integer
cs	int set_resolution (double newval)
java	int set_resolution (double newval)
py	def set_resolution (newval)
cmd	YGenericSensor target set_resolution newval

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

Parameters :

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

Returns :

YAPI_SUCCESS if the call succeeds.

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

genericsensor→set_signalRange()**YGenericSensor****genericsensor→setSignalRange()**

Changes the electric signal range used by the sensor.

js	function set_signalRange (newval)
nodejs	function set_signalRange (newval)
php	function set_signalRange (\$newval)
cpp	int set_signalRange (const string& newval)
m	-(int) setSignalRange : (NSString*) newval
pas	function set_signalRange (newval : string): integer
vb	function set_signalRange (ByVal newval As String) As Integer
cs	int set_signalRange (string newval)
java	int set_signalRange (String newval)
py	def set_signalRange (newval)
cmd	YGenericSensor target set_signalRange newval

Parameters :

newval a string corresponding to the electric signal range used by the sensor

Returns :

YAPI_SUCCESS if the call succeeds.

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

genericsensor→**set_unit()**
genericsensor→**setUnit()**

YGenericSensor

Changes the measuring unit for the measured value.

js	function set_unit (newval)
nodejs	function set_unit (newval)
php	function set_unit (\$newval)
cpp	int set_unit (const string& newval)
m	-(int) setUnit : (NSString*) newval
pas	function set_unit (newval : string): integer
vb	function set_unit (ByVal newval As String) As Integer
cs	int set_unit (string newval)
java	int set_unit (String newval)
py	def set_unit (newval)
cmd	YGenericSensor target set_unit newval

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

Parameters :

newval a string corresponding to the measuring unit for the measured value

Returns :

YAPI_SUCCESS if the call succeeds.

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

genericsensor→**set_userData()** **genericsensor**→**setUserData()**

YGenericSensor

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

js	function set_userData (data)
nodejs	function set_userData (data)
php	function set_userData (\$data)
cpp	void set_userData (void* data)
m	-(void) setUserData : (void*) data
pas	procedure set_userData (data : Tobject)
vb	procedure set_userData (ByVal data As Object)
cs	void set_userData (object data)
java	void set_userData (Object data)
py	def set_userData (data)

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

Parameters :

data any kind of object to be stored

genericsensor→set_valueRange() **genericsensor→setValueRange()**

YGenericSensor

Changes the physical value range measured by the sensor.

<code>js</code>	<code>function set_valueRange(newval)</code>
<code>nodejs</code>	<code>function set_valueRange(newval)</code>
<code>php</code>	<code>function set_valueRange(\$newval)</code>
<code>cpp</code>	<code>int set_valueRange(const string& newval)</code>
<code>m</code>	<code>-(int) setValueRange : (NSString*) newval</code>
<code>pas</code>	<code>function set_valueRange(newval: string): integer</code>
<code>vb</code>	<code>function set_valueRange(ByVal newval As String) As Integer</code>
<code>cs</code>	<code>int set_valueRange(string newval)</code>
<code>java</code>	<code>int set_valueRange(String newval)</code>
<code>py</code>	<code>def set_valueRange(newval)</code>
<code>cmd</code>	<code>YGenericSensor target set_valueRange newval</code>

The range change may have a side effect on the display resolution, as it may be adapted automatically.

Parameters :

newval a string corresponding to the physical value range measured by the sensor

Returns :

YAPI_SUCCESS if the call succeeds.

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

genericsensor→wait_async()**YGenericSensor**

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

```
js function wait_async( callback, context)
```

```
nodejs function wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the Javascript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

19.4. Recorded data sequence

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

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

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

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

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

YDataSet methods

dataset→`get_endTimeUTC()`

Returns the end time of the dataset, relative to the Jan 1, 1970.

dataset→`get_functionId()`

Returns the hardware identifier of the function that performed the measure, without reference to the module.

dataset→`get_hardwareId()`

Returns the unique hardware identifier of the function who performed the measures, in the form `SERIAL.FUNCTIONID`.

dataset→`get_measures()`

Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

dataset→`get_preview()`

Returns a condensed version of the measures that can retrieved in this YDataSet, as a list of YMeasure objects.

dataset→`get_progress()`

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

dataset→`get_startTimeUTC()`

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

dataset→`get_summary()`

Returns an YMeasure object which summarizes the whole DataSet.

dataset→`get_unit()`

Returns the measuring unit for the measured value.

dataset→loadMore()

Loads the the next block of measures from the dataLogger, and updates the progress indicator.

dataset→loadMore_async(callback, context)

Loads the the next block of measures from the dataLogger asynchronously.

dataset→**get_endTimeUTC()****YDataSet****dataset**→**endTimeUTC()**

Returns the end time of the dataset, relative to the Jan 1, 1970.

<code>js</code>	<code>function get_endTimeUTC()</code>
<code>nodejs</code>	<code>function get_endTimeUTC()</code>
<code>php</code>	<code>function get_endTimeUTC()</code>
<code>cpp</code>	<code>s64 get_endTimeUTC()</code>
<code>m</code>	<code>-(s64) endTimeUTC</code>
<code>pas</code>	<code>function get_endTimeUTC(): int64</code>
<code>vb</code>	<code>function get_endTimeUTC() As Long</code>
<code>cs</code>	<code>long get_endTimeUTC()</code>
<code>java</code>	<code>long get_endTimeUTC()</code>
<code>py</code>	<code>def get_endTimeUTC()</code>

When the YDataSet is created, the end time is the value passed in parameter to the `get_dataSet()` function. After the very first call to `loadMore()`, the end time is updated to reflect the timestamp of the last measure actually found in the dataLogger within the specified range.

Returns :

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

dataset→**get_functionId()****YDataSet****dataset**→**functionId()**

Returns the hardware identifier of the function that performed the measure, without reference to the module.

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

For example `temperature1`.

Returns :

a string that identifies the function (ex: `temperature1`)

dataset→**get_hardwareId()****YDataSet****dataset**→**hardwareId()**

Returns the unique hardware identifier of the function who performed the measures, in the form `SERIAL.FUNCTIONID`.

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

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example `THRMCPL1-123456.temperature1`)

Returns :

a string that uniquely identifies the function (ex: `THRMCPL1-123456.temperature1`)

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

dataset→**get_measures()****YDataSet****dataset**→**measures()**

Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

js	function get_measures ()
nodejs	function get_measures ()
php	function get_measures ()
cpp	vector<YMeasure> get_measures ()
m	-(NSMutableArray*) measures
pas	function get_measures (): TYMeasureArray
vb	function get_measures () As List
cs	List<YMeasure> get_measures ()
java	ArrayList<YMeasure> get_measures ()
py	def get_measures ()

Each item includes: - the start of the measure time interval - the end of the measure time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

Before calling this method, you should call `loadMore()` to load data from the device. You may have to call `loadMore()` several time until all rows are loaded, but you can start looking at available data rows before the load is complete.

The oldest measures are always loaded first, and the most recent measures will be loaded last. As a result, timestamps are normally sorted in ascending order within the measure table, unless there was an unexpected adjustment of the datalogger UTC clock.

Returns :

a table of records, where each record depicts the measured value for a given time interval

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

dataset→**get_preview()****YDataSet****dataset**→**preview()**

Returns a condensed version of the measures that can be retrieved in this YDataSet, as a list of YMeasure objects.

js	function get_preview ()
nodejs	function get_preview ()
php	function get_preview ()
c++	vector<YMeasure> get_preview ()
m	-(NSMutableArray*) preview
pas	function get_preview (): TYMeasureArray
vb	function get_preview () As List
cs	List<YMeasure> get_preview ()
java	ArrayList<YMeasure> get_preview ()
py	def get_preview ()

Each item includes: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This preview is available as soon as `loadMore()` has been called for the first time.

Returns :

a table of records, where each record depicts the measured values during a time interval

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

dataset→**get_progress()****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 ()
nodejs	function get_progress ()
php	function get_progress ()
cpp	int get_progress ()
m	-(int) progress
pas	function get_progress (): LongInt
vb	function get_progress () As Integer
cs	int get_progress ()
java	int get_progress ()
py	def get_progress ()

When the object is instanciated by `get_dataSet`, the progress is zero. Each time `loadMore()` is invoked, the progress is updated, to reach the value 100 only once all measures have been loaded.

Returns :

an integer in the range 0 to 100 (percentage of completion).

dataset→**get_startTimeUTC()****YDataSet****dataset**→**startTimeUTC()**

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

js	function get_startTimeUTC ()
nodejs	function get_startTimeUTC ()
php	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 ()
py	def 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.

<code>js</code>	<code>function get_summary()</code>
<code>nodejs</code>	<code>function get_summary()</code>
<code>php</code>	<code>function get_summary()</code>
<code>cpp</code>	<code>YMeasure get_summary()</code>
<code>m</code>	<code>-(YMeasure*) summary</code>
<code>pas</code>	<code>function get_summary(): TYMeasure</code>
<code>vb</code>	<code>function get_summary() As YMeasure</code>
<code>cs</code>	<code>YMeasure get_summary()</code>
<code>java</code>	<code>YMeasure get_summary()</code>
<code>py</code>	<code>def get_summary()</code>

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 ()
nodejs	function get_unit ()
php	function get_unit ()
cpp	string get_unit ()
m	-(NSString*) unit
pas	function get_unit (): string
vb	function get_unit () As String
cs	string get_unit ()
java	String get_unit ()
py	def get_unit ()

Returns :

a string that represents a physical unit.

On failure, throws an exception or returns Y_UNIT_INVALID.

dataset→loadMore()**YDataSet**

Loads the the next block of measures from the dataLogger, and updates the progress indicator.

js	function loadMore ()
nodejs	function loadMore ()
php	function loadMore ()
cpp	int loadMore ()
m	-(int) loadMore
pas	function loadMore (): LongInt
vb	function loadMore () As Integer
cs	int loadMore ()
java	int loadMore ()
py	def loadMore ()

Returns :

an integer in the range 0 to 100 (percentage of completion), or a negative error code in case of failure.

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

dataset→loadMore_async()**YDataSet**

Loads the the next block of measures from the dataLogger asynchronously.

```
js function loadMore_async( callback, context)
nodejs 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.

19.5. Measured value

YMeasure objects are used within the API to represent a value measured at a specified time. These objects are used in particular in conjunction with the YDataSet class.

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

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

YMeasure methods
measure→get_averageValue() Returns the average value observed during the time interval covered by this measure.
measure→get_endTimeUTC() Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).
measure→get_maxValue() Returns the largest value observed during the time interval covered by this measure.
measure→get_minValue() Returns the smallest value observed during the time interval covered by this measure.
measure→get_startTimeUTC() Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

measure→get_averageValue()**YMeasure****measure→averageValue()**

Returns the average value observed during the time interval covered by this measure.

js	function get_averageValue ()
nodejs	function get_averageValue ()
php	function get_averageValue ()
cpp	double get_averageValue ()
m	-(double) averageValue
pas	function get_averageValue (): double
vb	function get_averageValue () As Double
cs	double get_averageValue ()
java	double get_averageValue ()
py	def get_averageValue ()

Returns :

a floating-point number corresponding to the average value observed.

measure→**get_endTimeUTC()****YMeasure****measure**→**endTimeUTC()**

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

<code>js</code>	<code>function get_endTimeUTC()</code>
<code>nodejs</code>	<code>function get_endTimeUTC()</code>
<code>php</code>	<code>function get_endTimeUTC()</code>
<code>cpp</code>	<code>double get_endTimeUTC()</code>
<code>m</code>	<code>-(double) endTimeUTC</code>
<code>pas</code>	<code>function get_endTimeUTC(): double</code>
<code>vb</code>	<code>function get_endTimeUTC() As Double</code>
<code>cs</code>	<code>double get_endTimeUTC()</code>
<code>java</code>	<code>double get_endTimeUTC()</code>
<code>py</code>	<code>def get_endTimeUTC()</code>

When the recording rate is higher than 1 sample per second, the timestamp may have a fractional part.

Returns :

an floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the 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 ()
nodejs	function get_maxValue ()
php	function get_maxValue ()
cpp	double get_maxValue ()
m	-(double) maxValue
pas	function get_maxValue (): double
vb	function get_maxValue () As Double
cs	double get_maxValue ()
java	double get_maxValue ()
py	def get_maxValue ()

Returns :

a floating-point number corresponding to the largest value observed.

measure→**get_minValue()****YMeasure****measure**→**minValue()**

Returns the smallest value observed during the time interval covered by this measure.

js	function get_minValue ()
nodejs	function get_minValue ()
php	function get_minValue ()
cpp	double get_minValue ()
m	-(double) minValue
pas	function get_minValue (): double
vb	function get_minValue () As Double
cs	double get_minValue ()
java	double get_minValue ()
py	def get_minValue ()

Returns :

a floating-point number corresponding to the smallest value observed.

measure→get_startTimeUTC()
measure→startTimeUTC()

YMeasure

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

js	function get_startTimeUTC ()
nodejs	function get_startTimeUTC ()
php	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 ()
py	def get_startTimeUTC ()

When the recording rate is higher then 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.

19.6. Unformatted data sequence

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

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

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

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

YDataStream methods

datastream→get_averageValue()

Returns the average of all measures observed within this stream.

datastream→get_columnCount()

Returns the number of data columns present in this stream.

datastream→get_columnNames()

Returns the title (or meaning) of each data column present in this stream.

datastream→get_data(row, col)

Returns a single measure from the data stream, specified by its row and column index.

datastream→get_dataRows()

Returns the whole data set contained in the stream, as a bidimensional table of numbers.

datastream→get_dataSamplesIntervalMs()

Returns the number of milliseconds between two consecutive rows of this data stream.

datastream→get_duration()

Returns the approximate duration of this stream, in seconds.

datastream→get_maxValue()

Returns the largest measure observed within this stream.

datastream→get_minValue()

Returns the smallest measure observed within this stream.

datastream→get_rowCount()

Returns the number of data rows present in this stream.

datastream→get_runIndex()

Returns the run index of the data stream.

datastream→get_startTime()

Returns the relative start time of the data stream, measured in seconds.

datastream→get_startTimeUTC()

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

datastream→get_averageValue()**YDataStream****datastream→averageValue()**

Returns the average of all measures observed within this stream.

<code>js</code>	<code>function get_averageValue()</code>
<code>nodejs</code>	<code>function get_averageValue()</code>
<code>php</code>	<code>function get_averageValue()</code>
<code>cpp</code>	<code>double get_averageValue()</code>
<code>m</code>	<code>-(double) averageValue</code>
<code>pas</code>	<code>function get_averageValue(): double</code>
<code>vb</code>	<code>function get_averageValue() As Double</code>
<code>cs</code>	<code>double get_averageValue()</code>
<code>java</code>	<code>double get_averageValue()</code>
<code>py</code>	<code>def get_averageValue()</code>

If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.

Returns :

a floating-point number corresponding to the average value, or Y_DATA_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y_DATA_INVALID.

datastream→get_columnCount()**YDataStream****datastream→columnCount()**

Returns the number of data columns present in this stream.

<code>js</code>	<code>function get_columnCount()</code>
<code>nodejs</code>	<code>function get_columnCount()</code>
<code>php</code>	<code>function get_columnCount()</code>
<code>cpp</code>	<code>int get_columnCount()</code>
<code>m</code>	<code>-(int) columnCount</code>
<code>pas</code>	<code>function get_columnCount(): LongInt</code>
<code>vb</code>	<code>function get_columnCount() As Integer</code>
<code>cs</code>	<code>int get_columnCount()</code>
<code>java</code>	<code>int get_columnCount()</code>
<code>py</code>	<code>def get_columnCount()</code>

The meaning of the values present in each column can be obtained using the method `get_columnNames()`.

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

Returns :

an unsigned number corresponding to the number of columns.

On failure, throws an exception or returns zero.

datastream→**get_columnNames()****YDataStream****datastream**→**columnNames()**

Returns the title (or meaning) of each data column present in this stream.

<code>js</code>	<code>function get_columnNames()</code>
<code>nodejs</code>	<code>function get_columnNames()</code>
<code>php</code>	<code>function get_columnNames()</code>
<code>cpp</code>	<code>vector<string> get_columnNames()</code>
<code>m</code>	<code>-(NSMutableArray*) columnNames</code>
<code>pas</code>	<code>function get_columnNames(): TStringArray</code>
<code>vb</code>	<code>function get_columnNames() As List</code>
<code>cs</code>	<code>List<string> get_columnNames()</code>
<code>java</code>	<code>ArrayList<String> get_columnNames()</code>
<code>py</code>	<code>def get_columnNames()</code>

In most case, the title of the data column is the hardware identifier of the sensor that produced the data. For streams recorded at a lower recording rate, the dataLogger stores the min, average and max value during each measure interval into three columns with suffixes `_min`, `_avg` and `_max` respectively.

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

Returns :

a list containing as many strings as there are columns in the data stream.

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

datastream→**get_data()****YDataStream****datastream**→**data()**

Returns a single measure from the data stream, specified by its row and column index.

<code>js</code>	<code>function get_data(row, col)</code>
<code>nodejs</code>	<code>function get_data(row, col)</code>
<code>php</code>	<code>function get_data(\$row, \$col)</code>
<code>cpp</code>	<code>double get_data(int row, int col)</code>
<code>m</code>	<code>-(double) data : (int) row : (int) col</code>
<code>pas</code>	<code>function get_data(row: LongInt, col: LongInt): double</code>
<code>vb</code>	<code>function get_data() As Double</code>
<code>cs</code>	<code>double get_data(int row, int col)</code>
<code>java</code>	<code>double get_data(int row, int col)</code>
<code>py</code>	<code>def get_data(row, col)</code>

The meaning of the values present in each column can be obtained using the method `get_columnNames()`.

This method fetches the whole data stream from the device, if not yet done.

Parameters :

row row index
col column index

Returns :

a floating-point number

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

datastream→get_dataRows() datastream→dataRows()

YDataStream

Returns the whole data set contained in the stream, as a bidimensional table of numbers.

js	function get_dataRows ()
nodejs	function get_dataRows ()
php	function get_dataRows ()
cpp	vector< vector<double> > get_dataRows ()
m	-(NSMutableArray*) dataRows
pas	function get_dataRows (): TDoubleArrayArray
vb	function get_dataRows () As List
cs	List<List<double>> get_dataRows ()
java	ArrayList<ArrayList<Double>> get_dataRows ()
py	def get_dataRows ()

The meaning of the values present in each column can be obtained using the method `get_columnNames()`.

This method fetches the whole data stream from the device, if not yet done.

Returns :

a list containing as many elements as there are rows in the data stream. Each row itself is a list of floating-point numbers.

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

datastream→get_dataSamplesIntervalMs()**YDataStream****datastream→dataSamplesIntervalMs()**

Returns the number of milliseconds between two consecutive rows of this data stream.

js	function get_dataSamplesIntervalMs ()
nodejs	function get_dataSamplesIntervalMs ()
php	function get_dataSamplesIntervalMs ()
cpp	int get_dataSamplesIntervalMs ()
m	-(int) dataSamplesIntervalMs
pas	function get_dataSamplesIntervalMs (): LongInt
vb	function get_dataSamplesIntervalMs () As Integer
cs	int get_dataSamplesIntervalMs ()
java	int get_dataSamplesIntervalMs ()
py	def get_dataSamplesIntervalMs ()

By default, the data logger records one row per second, but the recording frequency can be changed for each device function

Returns :

an unsigned number corresponding to a number of milliseconds.

datastream→**get_duration()****YDataStream****datastream**→**duration()**

Returns the approximate duration of this stream, in seconds.

js	function get_duration ()
nodejs	function get_duration ()
php	function get_duration ()
cpp	int get_duration ()
m	-(int) duration
pas	function get_duration (): LongInt
vb	function get_duration () As Integer
cs	int get_duration ()
java	int get_duration ()
py	def get_duration ()

Returns :

the number of seconds covered by this stream.

On failure, throws an exception or returns Y_DURATION_INVALID.

datastream→get_maxValue()**YDataStream****datastream→maxValue()**

Returns the largest measure observed within this stream.

js	function get_maxValue ()
nodejs	function get_maxValue ()
php	function get_maxValue ()
cpp	double get_maxValue ()
m	-(double) maxValue
pas	function get_maxValue (): double
vb	function get_maxValue () As Double
cs	double get_maxValue ()
java	double get_maxValue ()
py	def get_maxValue ()

If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.

Returns :

a floating-point number corresponding to the largest value, or Y_DATA_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y_DATA_INVALID.

datastream→get_minValue()**YDataStream****datastream→minValue()**

Returns the smallest measure observed within this stream.

js	function get_minValue ()
nodejs	function get_minValue ()
php	function get_minValue ()
cpp	double get_minValue ()
m	-(double) minValue
pas	function get_minValue (): double
vb	function get_minValue () As Double
cs	double get_minValue ()
java	double get_minValue ()
py	def get_minValue ()

If the device uses a firmware older than version 13000, this method will always return Y_DATA_INVALID.

Returns :

a floating-point number corresponding to the smallest value, or Y_DATA_INVALID if the stream is not yet complete (still recording).

On failure, throws an exception or returns Y_DATA_INVALID.

datastream→get_rowCount() datastream→rowCount()

YDataStream

Returns the number of data rows present in this stream.

js	function get_rowCount ()
nodejs	function get_rowCount ()
php	function get_rowCount ()
cpp	int get_rowCount ()
m	-(int) rowCount
pas	function get_rowCount (): LongInt
vb	function get_rowCount () As Integer
cs	int get_rowCount ()
java	int get_rowCount ()
py	def get_rowCount ()

If the device uses a firmware older than version 13000, this method fetches the whole data stream from the device if not yet done, which can cause a little delay.

Returns :

an unsigned number corresponding to the number of rows.

On failure, throws an exception or returns zero.

datastream→get_runIndex()**YDataStream****datastream→runIndex()**

Returns the run index of the data stream.

js	function get_runIndex ()
nodejs	function get_runIndex ()
php	function get_runIndex ()
cpp	int get_runIndex ()
m	-(int) runIndex
pas	function get_runIndex (): LongInt
vb	function get_runIndex () As Integer
cs	int get_runIndex ()
java	int get_runIndex ()
py	def get_runIndex ()

A run can be made of multiple datastreams, for different time intervals.

Returns :

an unsigned number corresponding to the run index.

datastream→get_startTime()**YDataStream****datastream→startTime()**

Returns the relative start time of the data stream, measured in seconds.

js	function get_startTime ()
nodejs	function get_startTime ()
php	function get_startTime ()
cpp	int get_startTime ()
m	-(int) startTime
pas	function get_startTime (): LongInt
vb	function get_startTime () As Integer
cs	int get_startTime ()
java	int get_startTime ()
py	def get_startTime ()

For recent firmwares, the value is relative to the present time, which means the value is always negative. If the device uses a firmware older than version 13000, value is relative to the start of the time the device was powered on, and is always positive. If you need an absolute UTC timestamp, use `get_startTimeUTC()`.

Returns :

an unsigned number corresponding to the number of seconds between the start of the run and the beginning of this data stream.

datastream→get_startTimeUTC() datastream→startTimeUTC()

YDataStream

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

js	function get_startTimeUTC ()
nodejs	function get_startTimeUTC ()
php	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 ()
py	def get_startTimeUTC ()

If the UTC time was not set in the datalogger at the time of the recording of this data stream, this method returns 0.

Returns :

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

20. Troubleshooting

20.1. 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 Linux¹ archive, there are two rule examples which you can use as a basis.

Example 1: 51-yoctopuce.rules

This rule provides all the users with read and write access to the Yoctopuce USB peripherals. Access rights for all other peripherals are not modified. If this scenario suits you, you only need to copy the `"51-yoctopuce_all.rules"` file into the `/etc/udev/rules.d` directory and to restart your system.

```
# udev rules to allow write access to all users
# for Yoctopuce USB devices
SUBSYSTEM=="usb", ATTR{idVendor}=="24e0", MODE="0666"
```

Example 2: 51-yoctopuce_group.rules

This rule authorizes the `"yoctogroup"` group to have read and write access to Yoctopuce USB peripherals. Access rights for all other peripherals are not modified. If this scenario suits you, you

¹ <http://www.yoctopuce.com/FR/virtualhub.php>

only need to copy the "51-yoctopuce_group.rules" file into the "/etc/udev/rules.d" directory and restart your system.

```
# udev rules to allow write access to all users of "yoctogroup"
# for Yoctopuce USB devices
SUBSYSTEM=="usb", ATTR{idVendor}=="24e0", MODE="0664", GROUP="yoctogroup"
```

20.2. ARM Platforms: HF and EL

There are two main flavors of executable on ARM: HF (Hard Float) binaries, and EL (EABI Little Endian) binaries. These two families are not compatible at all. The compatibility of a given ARM platform with one of these two families depends on the hardware and on the OS build. ArmHF 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.

21. Characteristics

You can find below a summary of the main technical characteristics of your Yocto-4-20mA-Rx module.

Resolution	0.001 mA
Width	20 mm
Length	60 mm
Weight	10 g
USB connector	micro-B
Channels	2
Refresh rate	50 Hz
Input impedance	50 Ω
Accuracy	0.01 %
Supported Operating Systems	Windows, Linux (Intel + ARM), Mac OS X, Android
Drivers	no driver needed
API / SDK / Libraries (USB+TCP)	C++, Objective-C, C#, VB .NET, Delphi, Python, Java/Android
API / SDK / Libraries (TCP only)	Javascript, Node.js, PHP, Java
RoHS	yes
USB Vendor ID	0x24E0
USB Device ID	0x0037
Suggested enclosure	YoctoBox-Long-Thick-Black-Vents
Cables and enclosures	available separately

Index

A

Access 95
Accessories 3
Activating 96
Advanced 107
Android 95, 96
Assembly 15

B

Basic 63
Blueprint 281

C

C# 69
C++ 49, 54
calibrateFromPoints, YGenericSensor 197
Calibration 112
Callback 44
Characteristics 279
CheckLogicalName, YAPI 125
Command 27, 117
Compatibility 95
Concepts 17
Configuration 12
Connections 15

D

Data 110, 243, 262
DataLogger 21
Delphi 77
describe, YGenericSensor 198
describe, YModule 152
Description 27
DisableExceptions, YAPI 126
Distribution 16
download, YModule 153
Dynamic 83, 119

E

Elements 5, 6
EnableExceptions, YAPI 127
EnableUSBHost, YAPI 128
Error 36, 47, 54, 61, 68, 74, 81, 87, 93, 105
Event 107

F

Files 83
Filters 44
FindGenericSensor, YGenericSensor 195
FindModule, YModule 150
FirstGenericSensor, YGenericSensor 196

FirstModule, YModule 151
Fixing 15
FreeAPI, YAPI 129
functionCount, YModule 154
functionId, YModule 155
functionName, YModule 156
Functions 124
functionValue, YModule 157

G

General 17, 27, 124
GenericSensor 20, 28, 31, 39, 49, 57, 64, 70, 77, 83, 89, 98, 193
get_advertisedValue, YGenericSensor 199
get_averageValue, YDataStream 263
get_averageValue, YMeasure 256
get_beacon, YModule 158
get_columnCount, YDataStream 264
get_columnNames, YDataStream 265
get_currentRawValue, YGenericSensor 200
get_currentValue, YGenericSensor 201
get_data, YDataStream 266
get_dataRows, YDataStream 267
get_dataSamplesIntervalMs, YDataStream 268
get_duration, YDataStream 269
get_endTimeUTC, YDataSet 244
get_endTimeUTC, YMeasure 257
get_errorMessage, YGenericSensor 202
get_errorMessage, YModule 159
get_errorType, YGenericSensor 203
get_errorType, YModule 160
get_firmwareRelease, YModule 161
get_friendlyName, YGenericSensor 204
get_functionDescriptor, YGenericSensor 205
get_functionId, YDataSet 245
get_functionId, YGenericSensor 206
get_hardwareId, YDataSet 246
get_hardwareId, YGenericSensor 207
get_hardwareId, YModule 162
get_highestValue, YGenericSensor 208
get_icon2d, YModule 163
get_lastLogs, YModule 164
get_logFrequency, YGenericSensor 209
get_logicalName, YGenericSensor 210
get_logicalName, YModule 165
get_lowestValue, YGenericSensor 211
get_luminosity, YModule 166
get_maxValue, YDataStream 270
get_maxValue, YMeasure 258
get_measures, YDataSet 247
get_minValue, YDataStream 271
get_minValue, YMeasure 259
get_module, YGenericSensor 212
get_module_async, YGenericSensor 213

get_persistentSettings, YModule 167
get_preview, YDataSet 248
get_productId, YModule 168
get_productName, YModule 169
get_productRelease, YModule 170
get_progress, YDataSet 249
get_rebootCountdown, YModule 171
get_recordedData, YGenericSensor 214
get_reportFrequency, YGenericSensor 215
get_resolution, YGenericSensor 216
get_rowCount, YDataStream 272
get_runIndex, YDataStream 273
get_serialNumber, YModule 172
get_signalRange, YGenericSensor 217
get_signalUnit, YGenericSensor 218
get_signalValue, YGenericSensor 219
get_startTime, YDataStream 274
get_startTimeUTC, YDataSet 250
get_startTimeUTC, YDataStream 275
get_startTimeUTC, YMeasure 260
get_summary, YDataSet 251
get_unit, YDataSet 252
get_unit, YGenericSensor 220
get_upTime, YModule 173
get_usbBandwidth, YModule 174
get_usbCurrent, YModule 175
get_userData, YGenericSensor 221
get_userData, YModule 176
get_valueRange, YGenericSensor 222
GetAPIVersion, YAPI 130
GetTickCount, YAPI 131

H

HandleEvents, YAPI 132
High-level 123
HTTP 44, 117

I

InitAPI, YAPI 133
Installation 63, 69
Installing 27
Integration 54
Interface 148, 193
Introduction 1
isOnline, YGenericSensor 223
isOnline, YModule 177
isOnline_async, YGenericSensor 224
isOnline_async, YModule 178

J

Java 89
Javascript 31

L

Languages 117
Libraries 119
Library 54, 83, 122

Limitations 29
Linux 277
load, YGenericSensor 225
load, YModule 179
load_async, YGenericSensor 227
load_async, YModule 180
loadCalibrationPoints, YGenericSensor 226
loadMore, YDataSet 253
loadMore_async, YDataSet 254
Localization 11
Logger 110

M

Measured 256
Module 11, 18, 19, 28, 33, 41, 51, 59, 66, 72, 79, 85, 91, 100, 148

N

Native 22, 95
.NET 63
nextGenericSensor, YGenericSensor 228
nextModule, YModule 181

O

Objective-C 57
Optional 3

P

Paradigm 17
Platforms 278
Port 96
Porting 122
Power 16
Preparation 77
PreregisterHub, YAPI 134
Prerequisites 1
Presentation 5
Programming 17, 24, 107
Project 63, 69
Python 83

R

reboot, YModule 182
Recorded 243
Reference 123
RegisterDeviceArrivalCallback, YAPI 135
RegisterDeviceRemovalCallback, YAPI 136
RegisterHub, YAPI 137
RegisterHubDiscoveryCallback, YAPI 138
RegisterLogFunction, YAPI 139
registerTimedReportCallback, YGenericSensor 229
registerValueCallback, YGenericSensor 230
revertFromFlash, YModule 183

S

- saveToFlash, YModule 184
- SelectArchitecture, YAPI 140
- Sensor 112
- Sequence 243, 262
- Service 22
- set_beacon, YModule 185
- set_highestValue, YGenericSensor 231
- set_logFrequency, YGenericSensor 232
- set_logicalName, YGenericSensor 233
- set_logicalName, YModule 186
- set_lowestValue, YGenericSensor 234
- set_luminosity, YModule 187
- set_reportFrequency, YGenericSensor 235
- set_resolution, YGenericSensor 236
- set_signalRange, YGenericSensor 237
- set_unit, YGenericSensor 238
- set_usbBandwidth, YModule 188
- set_userData, YGenericSensor 239
- set_userData, YModule 189
- set_valueRange, YGenericSensor 240
- SetDelegate, YAPI 141
- SetTimeout, YAPI 142
- Sleep, YAPI 143
- Source 83
- Start 24

T

- Test 11
- triggerFirmwareUpdate, YModule 190
- Troubleshooting 277

U

- Unformatted 262
- UnregisterHub, YAPI 144
- Unsupported 117
- UpdateDeviceList, YAPI 145
- UpdateDeviceList_async, YAPI 146

V

- Value 256
- Variants 54
- VirtualHub 95, 117
- Visual 63, 69

W

- wait_async, YGenericSensor 241
- wait_async, YModule 191

Y

- YAPI 125-146
- yCheckLogicalName 125
- YDataSet 244-254
- YDataStream 263-275
- yDisableExceptions 126
- yEnableExceptions 127
- yEnableUSBHost 128
- yFindGenericSensor 195
- yFindModule 150
- yFirstGenericSensor 196
- yFirstModule 151
- yFreeAPI 129
- YGenericSensor 195-241
- yGetAPIVersion 130
- yGetTickCount 131
- yHandleEvents 132
- yInitAPI 133
- YMeasure 256-260
- YModule 150-191
- Yocto-4-20mA-Rx 18, 27, 31, 39, 49, 57, 63, 69, 77, 83, 89, 95
- yPreregisterHub 134
- yRegisterDeviceArrivalCallback 135
- yRegisterDeviceRemovalCallback 136
- yRegisterHub 137
- yRegisterHubDiscoveryCallback 138
- yRegisterLogFunction 139
- ySelectArchitecture 140
- ySetDelegate 141
- ySetTimeout 142
- ySleep 143
- yUnregisterHub 144
- yUpdateDeviceList 145
- yUpdateDeviceList_async 146