Yocto-RS485

User's guide

Table of contents

1. Introduction **1**

1.1. Safety Information 2

1.2. Environmental conditions 3

2. Presentation **5**

2.1. Common elements 5

2.2. Specific elements 6

2.3. Functional insulation 7

2.4. Optional accessories 7

3. First steps **9**

3.1. Prerequisites 9

3.2. Testing USB connectivity 10

3.3. Localization 11

3.4. Test of the module 11

3.5. Configuration 12

4. Assembly and connections **15**

4.1. Fixing 15

4.2. Connections 16

4.3. USB power distribution 16

5. The serial port **19**

5.1. Configurable parameters 19

5.2. Text-line based protocol 20

5.3. Text protocol delimited by STX and ETX 20

5.4. Frame-based binary protocol 20

5.5. MODBUS protocol 21

5.6. Wiegand Protocol 21

5.7. ASCII data stream 22

5.8. Binary data stream 22

5.9. Serial communication analyzer 22

6. Automatic measures **25**

6.1. Communication jobs ... 25
6.2. Tasks ... 26
6.3. Commands ... 28
6.4. The genericSensor functions ... 31
6.5. Configuration example ... 32

7. Programming, general concepts ... 35
7.1. Programming paradigm ... 35
7.2. The Yocto-RS485 module ... 37
7.3. Module ... 38
7.4. SerialPort ... 39
7.5. GenericSensor ... 41
7.6. DataLogger ... 42
7.7. Files ... 43
7.8. What interface: Native, DLL or Service ? ... 44
7.9. Programming, where to start? ... 46

8. Using the Yocto-RS485 in command line ... 47
8.1. Installing ... 47
8.2. Use: general description ... 47
8.3. Control of the SerialPort function ... 48
8.4. Control of the module part ... 48
8.5. Limitations ... 49

9. Using Yocto-RS485 with JavaScript / EcmaScript ... 51
9.1. Blocking I/O versus Asynchronous I/O in JavaScript ... 51
9.2. Using Yoctopuce library for JavaScript / EcmaScript 2017 ... 52
9.3. Control of the SerialPort function ... 54
9.4. Control of the module part ... 59
9.5. Error handling ... 61

10. Using Yocto-RS485 with PHP ... 63
10.1. Getting ready ... 63
10.2. Control of the SerialPort function ... 63
10.3. Control of the module part ... 66
10.4. HTTP callback API and NAT filters ... 68
10.5. Error handling ... 72

11. Using Yocto-RS485 with C++ ... 73
11.1. Control of the SerialPort function ... 73
11.2. Control of the module part ... 75
11.3. Error handling ... 78
11.4. Integration variants for the C++ Yoctopuce library ... 78

12. Using Yocto-RS485 with Objective-C ... 81
12.1. Control of the SerialPort function ... 81
12.2. Control of the module part ... 83
12.3. Error handling ... 86

13. Using Yocto-RS485 with Visual Basic .NET ... 87
13.1. Installation ... 87
13.2. Using the Yoctopuce API in a Visual Basic project ... 87

13.3. Control of the SerialPort function ... 88
13.4. Control of the module part ... 90
13.5. Error handling ... 92

14. Using Yocto-RS485 with C# ... 93
14.1. Installation ... 93
14.2. Using the Yoctopuce API in a Visual C# project ... 93
14.3. Control of the SerialPort function ... 94
14.4. Control of the module part ... 96
14.5. Error handling ... 98

15. Using the Yocto-RS485 with Universal Windows Platform ... 101
15.1. Blocking and asynchronous functions ... 101
15.2. Installation ... 102
15.3. Using the Yoctopuce API in a Visual Studio project ... 102
15.4. Control of the SerialPort function ... 103
15.5. A real example ... 104
15.6. Control of the module part ... 105
15.7. Error handling ... 107

16. Using Yocto-RS485 with Delphi ... 109
16.1. Preparation ... 109
16.2. Control of the SerialPort function ... 109
16.3. Control of the module part ... 112
16.4. Error handling ... 114

17. Using the Yocto-RS485 with Python ... 117
17.1. Source files ... 117
17.2. Dynamic library ... 117
17.3. Control of the SerialPort function ... 117
17.4. Control of the module part ... 119
17.5. Error handling ... 121

18. Using the Yocto-RS485 with Java ... 123
18.1. Getting ready ... 123
18.2. Control of the SerialPort function ... 123
18.3. Control of the module part ... 125
18.4. Error handling ... 128

19. Using the Yocto-RS485 with Android ... 129
19.1. Native access and VirtualHub ... 129
19.2. Getting ready ... 129
19.3. Compatibility ... 129
19.4. Activating the USB port under Android ... 130
19.5. Control of the SerialPort function ... 131
19.6. Control of the module part ... 134
19.7. Error handling ... 139

20. Using the Yocto-RS485 with LabVIEW ... 141
20.1. Architecture ... 141
20.2. Compatibility ... 142
20.3. Installation ... 142

20.4. Presentation of Yoctopuce VIs ... 147
20.5. Functioning and use of VIs ... 150
20.6. Using ... 152
20.7. Managing the data logger ... 154
20.8. Function list ... 155
20.9. A word on performances ... 156
20.10. A full example of a LabVIEW program ... 156
20.11. Differences from other Yoctopuce APIs ... 157

21. Using with unsupported languages ... 159
21.1. Command line ... 159
21.2. .NET Assembly ... 159
21.3. VirtualHub and HTTP GET ... 161
21.4. Using dynamic libraries ... 163
21.5. Porting the high level library ... 166

22. Advanced programming ... 167
22.1. Event programming ... 167
22.2. The data logger ... 170
22.3. Sensor calibration ... 172

23. Firmware Update ... 177
23.1. The VirtualHub or the YoctoHub ... 177
23.2. The command line library ... 177
23.3. The Android application Yocto-Firmware ... 177
23.4. Updating the firmware with the programming library ... 178
23.5. The "update" mode ... 180

24. High-level API Reference ... 181
24.1. Class YAPI ... 182
24.2. Class YModule ... 223
24.3. Class YSerialPort ... 300
24.4. Class YFiles ... 405
24.5. Class YGenericSensor ... 457
24.6. Class YDataLogger ... 549
24.7. Class YDataSet ... 608
24.8. Class YMeasure ... 641

25. Troubleshooting ... 647
25.1. Where to start? ... 647
25.2. Programming examples don't seem to work ... 647
25.3. Linux and USB ... 647
25.4. ARM Platforms: HF and EL ... 648
25.5. Powered module but invisible for the OS ... 648
25.6. Another process named xxx is already using yAPI ... 648
25.7. Disconnections, erratic behavior ... 648
25.8. Registering a VirtualHub disconnect an other one ... 649
25.9. Dropped commands ... 649
25.10. Damaged device ... 649

26. Characteristics ... 651

1. Introduction

The Yocto-RS485 is a USB 56x20mm module with a serial port at the RS485 standard (transmission in differential mode, enabling communications at more than a 1000m). It natively supports several serial protocols, including MODBUS. The module has a buffer memory enabling it to communicate asynchronously, if need be. The Yocto-RS485 can also work as an RS485 communication analyzer. In the opposite to most USB/serial adapters, it requires neither a driver and nor a virtual COM port.

On top of offering low level serial communications, the Yocto-RS485 can autonomously question and analyze the RS485 output of any appliance to then present the results in the manner of a Yoctopuce sensor. In other words, the Yocto-RS485 can transform any sensor equipped with an RS485 output into the software equivalent of a Yoctopuce sensor, data logger included.

An important characteristic of the Yocto-RS485 is to be an insulated module: the RS485 part of the module is electrically insulated from the USB part. Therefore, you can connect the module to devices powered by the mains, for example, without risking to destroy your computer, even if the appliance is not on the same phase.

Beware, the Yocto-RS485 is not a classic RS485 to USB adaptor: it does not create a virtual COM port, you cannot therefore use it with an application designed to use a COM port.

The Yocto-RS485 module

The Yocto-RS485 is not in itself a complete product. It is a component intended to be integrated into a solution used in laboratory equipments, or in industrial process-control equipments, or for similar applications in domestic and commercial environments. In order to use it, you must at least install it in a protective enclosure and connect it to a host computer.

Yoctopuce thanks you for buying this Yocto-RS485 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-RS485 is easy to install anywhere and easy to drive from a maximum of programming languages. If you are nevertheless disappointed with this module, or if you need additional information, do not hesitate to contact Yoctopuce support:

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

1.1. Safety Information

The Yocto-RS485 is designed to meet the requirements of IEC 61010-1:2010 safety standard. It does not create any serious hazards to the operator and surrounding area, even in single fault condition, as long as it is integrated and used according to the instructions contained in this documentation, and in this section in particular.

Protective enclosure

The Yocto-RS485 should not be used without a protective enclosure, because of the accessible bare electronic components. For optimal safety, it should be put into a non-metallic, non-inflammable enclosure, resistant to a mechanical stress level of 5 J. For instance, use a polycarbonate (e.g. LEXAN) enclosure rated IK08 with a IEC 60695-11-10 flammability rating of V-1 or better. Using a lower quality enclosure may require specific warnings for the operator and/or compromise conformity with the safety standard.

Maintenance

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

Identification

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

Identification label is integrated in the package label.

Application

The safety standard applied is intended to cover laboratory equipment, industrial process-control equipment and similar applications in residential or commercial environment. If you intend to use the Yocto-RS485 for another kind of application, you should check the safety regulations according to the standard applicable to your application.

In particular, the Yocto-RS485 is *not* certified for use in medical environments or for life-support applications.

Environment

The Yocto-RS485 is *not* certified for use in hazardous locations, explosive environments, or life-threatening applications. Environmental ratings are provided below.

IEC 61140 Protection Class III

The Yocto-RS485 has been designed to work with safety extra-low voltages only. Do not exceed voltages indicated in this manual, and never connect to the Yocto-RS485 terminal blocks any wire that could be connected to the mains.

1.2. Environmental conditions

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

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

The extended operating temperature range for the Yocto-RS485 is -30...85°C. This temperature range has been determined based on components manufacturer recommendations, and on controlled environment tests performed during a limited duration (1h). If you plan to use the Yocto-RS485 in harsh environments for a long period of time, we strongly advise you to run extensive tests before going to production.

2. Presentation

1: Micro-B USB socket	*5: DATA +*
2: Yocto-led	*6: DATA -*
3: Yocto-button	*7: Ground*
4: Activity leds	*8: Termination switch*

2.1. Common elements

All Yocto-modules share a number of common functionalities.

USB connector

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

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

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

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

Yocto-button

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

Yocto-led

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

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

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

Current sensor

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

Serial number

Each Yocto-module has a unique serial number assigned to it at the factory. For Yocto-RS485 modules, this number starts with RS485MK1. The module can be software driven using this serial number. The serial number cannot be modified.

Logical name

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

2.2. Specific elements

The connector

The RS485 input/output port of the Yocto-RS485 module is a terminal with DATA+, DATA-, and ground contacts. Cabling of an RS485 usually has these three wires, to which devices are cabled in parallel. Sometimes ground is left out from the bus, but this is a design weakness which works only when all the devices have a relatively similar ground. The EIA/TIA standard requires a ground wire.

The communication circuit is a safety extra low voltage (SELV) circuit. It should not be presented with voltages exceeding 20V, nor connected to mains circuits. The Yocto-RS485 endurance to surge and lightning transiants has not been tested. If you intend to use the Yocto-RS485 with wires longer than 30m or running outside, you should perform this testing yourself (see IEC 61000-4-5).

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

Activity leds

The Yocto-RS485 has two green leds reflecting the RS485 port activity, the first one for receiving and the second for transmitting.

The termination switch

To optimize transmission quality on a RS485 bus, it is recommended to terminate it at both ends with a resistance of 120Ω between D+ and D-. To avoid using an external component, the Yocto-RS485 is equipped with an embedded resistance that you can connect to the bus by simply setting its micro-switch to ON.

2.3. Functional insulation

The Yocto-RS485 is designed as two distinct electrical circuits, separated by a functional insulation. This insulation plays no role for the operator safety, since both circuits of the Yocto-RS485 work with safety extra low voltages (SELV) and are accessible without risk at any time. The insulation has been added in excess of safety requirements, to improve the reliability and the ease of use of the Yocto-RS485, allowing both circuits to work with different reference grounds.

Although the insulation plays no role for security, it has been designed according to the rules that would apply for a supplementary insulation on a secondary circuit. Its specifications of the functional insulation are as follows[3]:

- **Insulation voltage[4]:** 2.5kV
- **Clearance distance:** 4.5mm
- **Creepage distance:** 4.5mm
- **Material group:** Cat IIIa (FR4)

2.4. Optional accessories

The accessories below are not necessary to use the Yocto-RS485 module but might be useful depending on your project. These are mostly common products that you can buy from your favorite hacking store. To save you the tedious job of looking for them, most of them are also available on the Yoctopuce shop.

Screws and spacers

In order to mount the Yocto-RS485 module, you can put small screws in the 2.5mm assembly holes, with a screw head no larger than 4.5mm. The best way is to use threaded spacers, which you can then mount wherever you want. You can find more details on this topic in the chapter about assembly and connections.

Micro-USB hub

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

YoctoHub-Ethernet, YoctoHub-Wireless and YoctoHub-GSM

You can add network connectivity to your Yocto-RS485, thanks to the YoctoHub-Ethernet, the YoctoHub-Wireless and the YoctoHub-GSM which provides repectiveley Ethernet, WiFi and GSM connectivity. All of them can drive up to three devices and behave exactly like a regular computer running a *VirtualHub*.

[3] This description of the insulation applies to the latest revision of the product. Earlier revisions of the product might have smaller clearance and creepage distance. In order to get the clearance and creepage distance for an older device, contact Yoctopuce support and provide either the serial number of the device or the purchase reference.
[4] **Nominal value, not tested**

1.27mm (or 1.25mm) connectors

In case you wish to connect your Yocto-RS485 to a Micro-hub USB or a YoctoHub without using a bulky USB connector, you can use the four 1.27mm pads just behind the USB connector. There are two options.

You can mount the Yocto-RS485 directly on the hub using screw and spacers, and connect it using 1.27mm board-to-board connectors. To prevent shortcuts, it is best to solder the female connector on the hub and the male connector on the Yocto-RS485.

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

Enclosure

Your Yocto-RS485 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 [5]. The suggested enclosure model for your Yocto-RS485 is the YoctoBox-Long-Thick-Black.

You can install your Yocto-RS485 in an optional enclosure

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

3. First steps

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

3.1. Prerequisites

In order to use your Yocto-RS485 module, you should have the following items at hand.

A computer

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

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

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

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

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

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

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

A USB 2.0 cable, type A-micro B

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

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

To connect your Yocto-RS485 module to a computer, you need a USB 2.0 cable of type A-micro B. The price of this cable may vary a lot depending on the source, look for it under the name *USB 2.0 A to micro B Data cable*. Make sure not to buy a simple USB charging cable without data connectivity. The correct type of cable is available on the Yoctopuce shop.

You must plug in your Yocto-RS485 module with a USB 2.0 cable of type A - micro B

If you insert a USB hub between the computer and the Yocto-RS485 module, make sure to take into account the USB current limits. If you do not, be prepared to face unstable behaviors and unpredictable failures. You can find more details on this topic in the chapter about assembly and connections.

3.2. Testing USB connectivity

At this point, your Yocto-RS485 should be connected to your computer, which should have recognized it. It is time to make it work.

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

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

Module list as displayed in your web bowser

3.3. Localization

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

3.4. Test of the module

The first item to check is that your module is working well: click on the serial number corresponding to your module. This displays a window summarizing the properties of your Yocto-RS485.

Properties of the Yocto-RS485 module

This window allows you, among other things, to play with your module to check how it is working. You can find there a simplified terminal emulator enabling you to test the communications with your device.

As soon as the window opens, the last 4KB of messages found in the device memory are displayed. As long as messages are exchanged, they will be appended to the list. By typing a keyword next to the magnifier icon, you can restrict the display to messages including the selected keyword. If needed, you can temporarily pause the display update using button **pause**. The user interface can handle hundredths of messages. After a few thousands, the Web browser is likely to start slowing down, but you can always use the **clear** button to empty the display history and the device memory.

Each message is prepended by a timestamp, relative to the first message, and by the direction of the transmission - send or receive. To make it easier to read, one of the direction is highlighted as well. The message timestamps are recorded directly by the device, at the exact time when the message is detected. They are therefore quite precise, with a resolution of a millisecond.

Regardless of the selected protocol type, you can switch the display between ASCII and hex mode using the button **view hex** / **view ASCII**. Be aware however that when the capture is made using a text-based protocol (Line-based ASCII protocol, STX/ETX-based ASCII protocol or generic ASCII

stream), all non-textual control codes will be automatically filtered and will therefore not appear, even in hex mode. So use a binary mode protocol if you need to check the control codes.

If you need to study or to compare a communication involving many messages, use the button **export** to open the whole set of messages in a larger window. The exported view includes simultaneously hex codes, with 16 bytes per line, and the corresponding ASCII characters, formatted as text lines to facilitate reading. This view can be saved into a stand-alone HTML file if needed, using the Save button, to be reopened later using any Web browser.

3.5. Configuration

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

Yocto-RS485 module configuration.

Firmware

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

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

Logical name of the module

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

[5] www.yoctopuce.com/EN/virtualhub.php
[6] More information available in the virtual hub documentation

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.

Serial port configuration

You can configure the workings of the serial port in this window. You can select the speed, the encoding, the parity, the number of stop bits, and so on.

You can also select the protocol you want to use on the serial line. You can find more details about these different protocols in the chapter entitled *5. The serial port*.

4. Assembly and connections

This chapter provides important information regarding the use of the Yocto-RS485 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-RS485 module contains 2.5mm assembly holes. You can use these holes for screws. The screw head diameter must not be larger than 4.5mm or they will damage the module circuits. Make sure that the lower surface of the module is not in contact with the support. We recommend using spacers, but other methods are possible. Nothing prevents you from fixing the module with a glue gun; it will not be good-looking, but it will hold.

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

4.2. Connections

The RS485 standard consists in driving the DATA+ and DATA- lines in differential mode, enabling it to route the signal on very long distances, sometimes more than a 1000 meters. the Yocto-RS485 communicates in half-duplex: each device speaks one after the other on the same line. RS485 wiring usually has three wires: DATA+, DATA-, and ground. Sometimes ground is missing from the bus but this is a design weakness which works only when all the devices have a rather similar ground. The EIS/TIA-485 standard requires a ground wire.

The Yocto-RS485 is an insulated module: do not artificially connect the RS485 bus ground to the USB bus ground, you would lose the the benefits from the insulation protecting your computer.

Direct connection to an RS485 device

The termination

In the opposite to a classic serial communication which works only from point to point, RS485 devices can be organized in a bus. Each end of the bus must end with a 120Ω resistance. To avoid using an external component, such a termination resistance is embedded into the Yocto-RS485. You can optionally enable it with a micro-switch located right next to the terminal.

Organization as a bus: beware of the termination.

4.3. USB power distribution

Although USB means *Universal Serial BUS*, USB devices are not physically organized as a flat bus but as a tree, using point-to-point connections. This has consequences on power distribution: to

make it simple, every USB port must supply power to all devices directly or indirectly connected to it. And USB puts some limits.

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

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

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

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

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

4.4. Electromagnetic compatibility (EMI)

Connection methods to integrate the Yocto-RS485 obviously have an impact on the system overall electromagnetic emissions, and therefore also impact the conformity with international standards.

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

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

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

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

5. The serial port

In the opposite to classic RS232 adapters, the Yocto-RS485 serial port is not a simple gateway to a virtual COM port. It is based on an active communication management by the module and offers a full programming interface like for all Yoctopuce modules. In particular,

- it does not require a driver
- its identifier remains the same when one changes the USB port on which it is connected
- it does not install any virtual COM port
- it can recognize some common message formats to ease software access
- it has circular input and output buffers, both of a size of 16 KB
- it permanently stores the data it receives, even when the computer is rebooting
- it can be accessed in parallel by several applications (data read by one application remain available for another application)
- it can measure the precise delay between each received byte to correctly decode binary protocols such as MODBUS

Thanks to these functions, you can use the Yocto-RS485 to, for example, perform serial communications from a simple command line or from HTTP request on a REST interface, without risking to lose messages.

The Yocto-RS485 RS485 port follows ANSI/TIA/EIA-485-A-98 and ISO 8482:1987(E) standards, and provides a ±15 kV ESD protection on RS-485 input/output pins.

5.1. Configurable parameters

The Yocto-RS485 serial port can generate communication speeds from 110 bits/s to 250'000 Kbits/s. You can configure it to use 7 or 8 bits of data, with or without parity (even or odd), with 1 or 2 stop bits.[1]

You can also configure in the module the protocol family to be used on the serial port. This allows the module to make a pre-analysis of the data directly when it receives them and to optimize the information exchange with the application code, in particular to signal the reception of new data at the most appropriate time (that is when a full message is received). Details of supported protocol families are available in the following sections.

[1] The only combination which is not supported is 7-N-1 (7 bits, without parity, only one stop bit) but this combination is almost never used.

5.2. Text-line based protocol

Called **Line-based ASCII protocol** in the configuration interface, it is a very common family for measuring tools. The host machine sends its configuration commands in the shape of commands ending by a line feed. The measuring tool sends its measures and its receipts as lines of text as well. Among the machines using this kind of protocol, you can find:

- GSM or other communication modules (AT commands)
- navigation tools following the NMEA standard (including GPS)
- Dylos particle sensors
- Geiger counters from Mazur Instruments
- and so on.

The most useful API functions in this working mode are:

- **reset()**, to empty the incoming buffer
- **get_lastMessage()** to obtain the latest line (latest measure) transmitted
- **queryMessage()** to send a command and wait for an answer for a given time set in advance.
- **writeLine()** and **readLine()**, to send commands and read answers manually, line by line (non-blocking functions)

In line mode, if you register a value notification callback, it is called after each newly received message.

5.3. Text protocol delimited by STX and ETX

Called **STX/ETX-based ASCII protocol** in the configuration interface, it is specific encoding used by some measuring tools. Message are sent in clear text, framed by a STX code at the beginning and an ETX code at the end. Proprietary binary codes may follow after the ETX code, but they will be ignored by the interface to make the decoding easier.

The most useful API functions in this working mode are:

- **reset()**, to empty the incoming buffer
- **writeStxEtx()**, to send a command manually, including the STX and ETX markers
- **get_lastMessage()** to obtain the latest line (latest measure) transmitted
- **queryMessage()** to send a command and wait for an answer for a given time set in advance.

In line mode, if you register a value notification callback, it is called after each newly received message.

5.4. Frame-based binary protocol

Called**Frame-based binary protocol** in the configuration interface, this family includes all proprietary protocols based on exchanging binary messages (non-textual).The MODBUS RTU protocol is a particular case which is explicitly managed (see below). But any other variant of binary frame exchanges can be used here. The Yocto-RS485 is able to separate the distinct received messages thanks to a measure of the delay in the successive byte reception. When you select a protocol based on binary frames, you can specify the space delimiting the separation between two frames.

Note that binary frames are limited to 256 bytes. Above this limit, a new frame is created for the later part of the message. The timestamp of the frames make it possible to recognize that the next frame is the continuation of the previous one.

If your binary protocol does not specify any constraint on the space between frames and space between the characters of a frame, you had better use the "Binary data stream" family below.

The most useful API functions in this working mode are:

- **reset()**, to empty the incoming buffer
- **get_lastMessage()** to obtain the hexadecimal representation of the latest frame received
- **writeHex()** to send a frame, given in hexadecimal
- **readMessages()** to look for a frame corresponding to a given regular expression
- **readHex()** to read a given number of bytes in hexadecimal (independently of frame distinctions)
- **queryHex()** to send an hexadecimal command and wait for an answer for a given time set in advance.

In binary frame mode, if you register a value notification callback, it is called for each newly received frame.

5.5. MODBUS protocol

The MODBUS protocol is much used in the industry and for monitoring the technical infrastructure of buildings. The protocol has two variants: the **MODBUS ASCII** mode where messages are exchanged as lines of hexadecimal code, and the **MODBUS RTU** mode where messages are exchanged directly as binary frames. To dialog with a MODBUS device, you imperatively must use the same mode as configured in the device. In theory, all the devices conforming to the standard must support the MODBUS RTU mode.

MODBUS messages correspond to relatively simple operations to read and write in binary registers (called bits or coils) and to 16 bit words. The host systematically initiates the exchange, to which the "slave" answers. The Yocto-RS485 transparently manages the ASCII and RTU modes and computes by itself the validation bytes (LRC and CRC) specified in the MODBUS protocol. The most useful API functions in MODBUS mode are:

- **modbusReadBits()** to read one or several status bits ("Read coils" MODBUS function)
- **modbusReadInputBits()** to read one or several input bits ("Read input bits" MODBUS function)
- **modbusReadRegisters()** to read one or several status/holding registers ("Read holding registers" MODBUS function)
- **modbusReadInputRegisters()** to read one or several input registers ("Read input registers" MODBUS function)
- **modbusWriteBit()** to modify a status/holding bit ("Write single coil" MODBUS function)
- **modbusWriteBits()** to modify several status/holding bits ("Write multiple coils" MODBUS function)
- **modbusWriteRegister()** to modify a single status/holding register ("Write single registers" MODBUS function)
- **modbusWriteRegisters()** to modify several status/holding registers ("Write multiple registers" MODBUS function)
- **modbusWriteAndReadRegisters()** to modify several registers and then directly read several status/holding registers ("Read/Write Multiple Registers" MODBUS function)
- **queryMODBUS()** to send an arbitrary MODBUS request and wait for the answer

In MODBUS mode, if you register a value notification callback, it is called each time an answer is received.

5.6. Wiegand Protocol

The Wiegand protocol is mainly used in access control systems (magnetic cards, RFID readers). There are many Wiegand variants, but all of them end up in sending a sequence of bits to identify an access tag. The Yocto-RS485 can decode Wiegand messages in two ways: either by sending the bit sequence "as is", in ASCII, as a string of zero and ones, or by combining the bits into bytes in order to facilitate the decoding. In order to allow the bits to be properly combined into bytes, the only thing you have to specify is the number of parity bits that needs to be taken out at the head of the message.

The most useful API functions in Wiegand mode are:

- **reset()**, to empty the incoming buffer
- **get_lastMessage()**, to read the ID of the last access tag seen
- **reset()** to forget all access tag IDs previously seen

5.7. ASCII data stream

Called **Generic ASCII stream** in the configuration interface, it is the most primitive text communication variant, similar to a file access. As the Yocto-RS485 has a 16KB read buffer, it is even possible to move the read position pointer freely within this window. Note that the read position pointer is specific to each application: if two applications read access the serial port through the network simultaneously, moving the read position pointer of one application does not have any impact on data availability for the other application.

The most useful API functions to to work with an ASCII data stream are:

- **reset()**, to empty the incoming buffer
- **writeStr()** to send a character string
- **readStr()** to read a character string
- **read_seek()** to move the read position pointer in the buffer
- **get_rxCount()** to know the position of the end of the receiving buffer

In stream mode, if you register a value notification callback, it is called each time a byte is received.

5.8. Binary data stream

Called **Generic byte stream** in the configuration interface, it is the binary equivalent of the ASCII data stream. You access it as you would a binary file, with the possibility to move the read position pointer freely inside the 16KB read buffer. Note that the read position pointer is specific to each application: if two applications read access the serial port through the network simultaneously, moving the read position pointer of one application does not have any impact on data availability for the other application.

The most useful API functions to to work with a binary data stream are:

- **reset()**, to empty the incoming buffer
- **writeHex()** to send a byte string (given a hexadecimal string)
- **readHex()** to read a byte string (given as a hexadecimal string)
- **read_seek()** to move the read position pointer within the buffer
- **get_rxCount()** to know the position of the end of the receiving buffer

In stream mode, if you register a value notification callback, it is called each time a byte is received.

5.9. Serial communication analyzer

You can use the Yocto-RS485 as a serial protocol analyzer by connecting it on the cable that directly connects two devices communicating via a serial protocol. In this specific mode, the Yocto-RS485 transmitting signals are not wired (ensuring trouble-free operations), and the TD and RD lines to be monitored are two receiving signals on the Yocto-RS485. It is then able to read the traffic going through both ways on the serial cable, identifying the direction of the communication.

Wiring to use the Yocto-RS485 in analyzer mode

If you want, you can buy a ready-to-use adapter on Yoctopuce shop with a female DB9 socket and a male DB9 plug. You will find it under the name RS232-Snooping-Adapter[2].

[2] http://www.yoctopuce.com/EN/products/accessories-and-connectors/rs232-snooping-adapter

6. Automatic measures

On top of offering the means to perform low level serial communications, the Yocto-RS485 can work at a superior abstraction level. It can autonomously question a device through the serial port and present the values read as measures, in the same manner as all the Yoctopuce sensors. This includes the possibility to store the measures on the internal flash memory (data logger). Potentially, this enables you to transform any device equipped with a serial port into a native Yoctopuce sensor, with all the advantages this brings in terms of ease for software integration.

The Yocto-RS485 can automatically send and receive data on the serial port.

6.1. Communication jobs

The Yocto-RS485 contains a file system in which you can store jobs, which are in fact simple text files in the JSON format. A job describes write and read actions to be performed on the serial port. In the VirtualHub interface, the window describing the Yocto-RS485 properties allows you to select which job must be run, while the configuration window enables you to select which job must be run when the module starts. The Yocto-RS485 runs only one job at a time, but a job can perform actions in parallel.

Job structure

A job is essentially a set of tasks which are independent from one another. Each task can send data on the serial port and/or react to the arrival of data on the serial port.

Job definition and management

You can define jobs with the VirtualHub, in the Yocto-RS485 configuration window. Click on the **manage files** button and a window containing the list of defined jobs appears.

Job management window

This window enables you to select which job to run, to edit, or to delete. It also allows you to define a new job, either with the help of an interface or by directly uploading it on the module file system. To create a new job, click on **define a new job**. This opens the job creation window.

Job creation window

As a job is only a set of tasks, this window only allows you to give a name to the job and to manage the tasks it contains.

Creating a job by software

While there is no explicit API to define a job by software, a job is only a text file put in the file system of the Yocto-RS485. To configure the Yocto-RS485 by software, you simply need to upload the correct file on the Yocto-RS485 with the *YFiles* class and to program its running with the *selectJob()* or *set_startupJob()* functions of the *YSerialPort* class. The easiest way to create a job file without risking to make an error consists in using the VirtualHub to configure the desired job on a Yocto-RS485 module, and then to download the corresponding file.

6.2. Tasks

Each task is a simple list of commands to run sequentially: sending data on the serial port, waiting, reading data, and so on. There are two types of tasks: reactive tasks and periodic tasks.

Reactive tasks

A reactive task is triggered by the device connected to the Yocto-RS485: the task is automatically run as soon as data corresponding to predefined patterns appear on the port. Most often, a task simply consists in interpreting these data and in assigning them to one or several *genericSensor* functions available on the Yocto-RS485. Reactive tasks are particularly useful to interface devices that send a continuous flow of measures on their serial port. If the module detects data with a pattern corresponding to two distinct tasks, both tasks are run in parallel.

The VirtualHub allows you to easily create a good number of reactive tasks, such as:

- Reading a numerical value in numeral or hexadecimal format
- Reading GPS data in NMEA format
- Reading a list of data in the CVS format
- Detecting a MODBUS write operation
- Detecting a MODBUS read operation

But you can also define personalized tasks by typing the task commands directly.

Reactive task creation interface

You can assign data that are read to any Yocto-RS485 *genericSensor* functions. From the developer stand point, the device that is connected to the Yocto-RS485 through its serial port appears like any usual Yoctopuce sensor. All the normal Yoctopuce sensor features (callbacks, data logger, averaging, and so on) are then available without any additional effort.

Beware, the serial protocol defined in the Yocto-RS485 configuration must correspond to the needs of the job. For instant, you cannot detect a MODBUS transaction if the Yocto-RS485 is configured in *line-based ASCII* mode.

Periodic tasks

A periodic task is a task that is run at a regular interval, at the Yocto-RS485 initiative. They are generally used to send commands to the device connected to the Yocto-RS485. Here again, the VirtualHub allows you to easily define a number of common tasks:

- Sending text data
- Sending binary data

- Performing a MODBUS write operation
- Performing a MODBUS read operation
- Performing a read operation through a SCPI command

You can also define a task manually, one command after the other, or start by using a predefined task as above and edit it afterwards to add commands.

You can also assign data read in a periodic task to the Yocto-RS485 *genericSensor* functions. Beware, the serial protocol defined in the Yocto-RS485 configuration must correspond to the needs of the job: for example, you cannot detect a MODBUS transaction if the Yocto-RS485 is configured in *line-based ASCII* mode.

Periodic task creation interface

Although periodic tasks are designed to be run at regular intervals, you can define a "periodic" task that is run only once. Periodic tasks are run in the order in which they are defined. You can therefore define a job containing a first non recurrent task to configure your device, and a second task, recurrent, to question it in a loop.

You can mix periodic and reactive tasks in a same job. You must however pay careful attention to their triggering events in order to prevent them from perturbing each other. The Yocto-RS485 always waits until the end of a periodic task before running the next one. However, reactive tasks can be triggered at any time, even in parallel to a periodic task.

6.3. Commands

You can use the following commands in a (periodic or reactive) task:

EXPECT

The *expect* command waits for the data corresponding to a given pattern to appear on the serial line. If the module is configured in binary mode, the correspondence is established on a hexadecimal representation of the binary data.

The *expect* command takes a character string as argument. We support some regular expressions:

- **.** (dot) corresponds to any character
- **[]** defines a union, for example **[123a-z]** corresponds to any character among **1**...**3**,**a**...**z**
- **[^]** defines an exclusion, for example **[^,]** corresponds to any character except the comma.
- ***** allows repetition of the latest correspondence zero, once, or several times: **.*** corresponds to all the characters until the end of the line
- **+** allows repetition of the latest correspondence once or several times. For example, **[0-9]+** corresponds to one or several digits.

Special expressions can decode and assign values to one of the device *genericSensor*:

- **($x:INT)** allows you to recognize an integer value (base 10) and assigned it to the corresponding *genericSensorX* function. For example, **{$3:INT}** allows you to recognize an integer value and to assign it to the *genericSensor3* function.
- **($x:FLOAT)** allows you to recognize a decimal value (floating point number) that is assigned to the *genericSensorX*. function. Scientific notation (for example **1.25e-1**) is supported.
- **($x:DDM)** allows you to recognize a decimal value in degree-minute-decimal format as used by NMEA standard
- **($x:BYTE)** allows you to recognize a value between 0 and 255 represented in hexadecimal, as for instance when using binary protocols. If the value is in the range -128...127, you should use **($x:SBYTE)** instead (where the S stands for *signed byte*. The decoded value is assigned to the *genericSensorX*.
- **($x:WORD)** or **($x:SWORD)** allows you similarly to decode a 16-bit hexadecimal value, unsigned or signed, and assign the result to the *genericSensorX*. The byte order is expected to be the natural written byte order (*big-endian*), i.e. the most significant byte first like in 0104 to represent 260.
- **($x:WORDL)** or **($x:SWORDL)** allows you to decode in the same way a 16-bit hexadecimal unsigned or signed value with the bytes in *little-endian* order, i.e. with the least significant byte first (eg. 0401 to represent 260).
- **($x:DWORD)** or **($x:SDWORD)** allows you to decode in the same way a 32-bit hexadecimal unsigned or signed value in *big-endian* byte order.
- **($x:DWORDL)** ou **($x:SDWORDL)** allows you to decode in the same way a 32-bit hexadecimal unsigned or signed value in *little-endian* byte order.
- **($x:DWORDX)** ou **($x:SDWORDX)** allows you to decode in the same way a 32-bit hexadecimal unsigned or signed value in *mixed-endian* byte order: two 16-bit words, each of them in big-endian, but the least significant 16-bit word is put first and the most significant word is put second.
- **($x:HEX)** allows you to decode a variable length hexadecimal number (1 to 4 byes in big-endian order), that is assigned to the *genericSensorX*.
- **($x:FLOAT16B)** and **($x:FLOAT16L)** allows you to decode a 16-bit floating point number encoded in hexadecimal according to IEEE 754, using *big-endian* or *little-endian* byte order.
- **($x:FLOAT16D)** allows you to decode a floating point number encoded in hexadecimal on two bytes, with the first byte representing the mantissa and the second byte representing the decimal signed exponent.
- **($x:FLOAT32B)** and **($x:FLOAT32L)** allows you to decode a 32-bit floating point number encoded in hexadecimal according to IEEE 754, using *big-endian* or *little-endian* byte order.
- **($x:FLOAT32X)** allows you to decode a 32-bit floating point number encoded in hexadecimal according to IEEE 754, using *mixed-endian* byte order, i.e. two 16-bit words, each of them in big-endian, but the least significant 16-bit word is put first and the most significant word is put second.

As the internal representation of floating point numbers in Yoctopuce devices is limited to 3 decimals, it is possible to change the magnitude of floating point numbers decoded by FLOAT, FLOAT16 and FLOAT32 expressions by prefixing them with a *M* to count in thousandths, or a *U* to count in

millionths (U like *micro*), or a *N* to count in billionths (N like *nano*). For instance, when the value **1.3e-6** is recognized with expression **($1:UFLOAT)**, the value assigned to *genericSensor1* is 1.3.

COMPUTE

The *compute* command can be used to perform intermediary computations based one previously parsed values. For instance the following code recognizes an integer and stores in a variable $t, then uses *compute* with this variable to convert it to Â°F and place the result in genericSensor1.

```
expect  ($t:WORD)
compute $1 = 32 + ($t * 9) / 5
```

You can use quite sophisticated mathematical expressions. Most usual mathematical operators are available, with the following precedence order:

`**`	raise to the power
`~ + - not`	complement, unary plus/minus, logical not
`* / % //`	multiply, divide, integer modulo, integer divide
`+ -`	add, subtract
`>> <<`	bitwise shift right and left
`&`	bitwise AND
`| ^`	bitwise OR, XOR
`< <= >= >`	compare
`== <> !=`	test if equal or different
`and`	logical AND
`or`	logical OR

For convenience, some alternative operator symbols can be used:

`div mod`	can be used instead of / and %
`! && ||`	can be used instead of `not`, `and`, `or`

Comparison and logical operators can be used together with the conditional evaluation operator:

```
compute "($temp &gt; 0 ? log($temp) : -9999)"
```

Standard math constants and functions are available as well:

`pi e`	the universal constants
`cos sin tan`	trigonometric functions
`acos asin atan atan2`	inverse trigonometric functions
`cosh sinh tanh`	hyperbolic functions
`exp log log10 pow sqr sqrt`	power and logarithmic functions
`ceil floor round frac fmod`	rounding functions
`fabs min max isnan isinf`	ranging functions

Computations are made on 32bit floating point numbers. Bitwise operators (| & >> etc.) are made on 32bit integers.

ASSERT

The *assert* command can be used to check if precondition is met before continuing a task. It takes as argument an arithmetic expression, and stops the task execution if the expression result is FALSE.

Similarly to the *compute* command, if the expression includes a syntax error or refers to an undefined variable, the task will be stopped as well, with an error message in the device logs. It is however possible to verify if a variable is defined without generating an error message using the special function `isset()`:

```
assert    !isset($init_done)
```

```
writeline :RESET
compute   $init_done = 1
```

WAIT

The *wait* command waits for a given number of milliseconds before running the following command.

LOG

The *log* command displays a character string in the logs of the Yocto-RS485.

WRITE

The *write* command sends a character string as is on the serial line. The string is sent without additional carriage return.

WRITELINE

The *writeLine* command sends a character string on the serial line, followed by a line break (CR-LF).

WRITEHEX

The *writeHex* command sends a binary message on the serial line. The parameter is the hexadecimal representation of the message to be sent (lower and upper case are supported).

WRITEMODBUS

The *writeModbus* command sends a MODBUS command. The parameter is a hexadecimal representation of the command to be sent, without checksum. For example:

- `03050006ff00` = Slave 3, Set Coil (05), relay 7 (0006), activate (0xff00)
- `030600021000` = Slave 3, Set Register (06), register 30003 (0002), value 4096 (0x1000)

SETRTS

The *setRTS* command, available only on the Yocto-Serial and on the Yocto-RS232 when hardware flux control is disabled, makes it possible to manually drive the RTS line from within a task.

SETSS

The *setSS* command, available only on the Yocto-SPI when frame-based transmission is not active, makes it possible to manually drive the SS line from within a task.

SETPOWER

The *setPower* command, available only on the Yocto-Serial and on the Yocto-SPI, makes it possible to automatically drive the power supply output from within a task, for instance to power on/off an external sensor.

If, for a reason or another, a command generates an error, you can find the traces of this error in the logs of the Yocto-RS485.

6.4. The genericSensor functions

9 *genericSensor* functions are available in the Yocto-RS485. Jobs running on the module can freely assign them values. You can access these *genericSensor* functions from the Yoctopuce API with the *YGenericSensor* class. You can also configure these functions to tailor their behavior depending on the nature of the reported values.

genericSensor configuration window

Unit

You can define the measuring system in which the value stored by the genericSensor is specified.

Resolution

You can define the resolution in which the value reported by the genericSensor must be represented.

Mapping

You can automatically apply a linear transformation to the values stored in a *genericSensor*. Indeed, some devices do not directly provide a physical quantity on their serial output. Let us imagine a voltmeter transmitting values between 0 and 65535 for measures between 0 and 10V. You can have the *genericSensor* function automatically perform an inverse conversion as illustrated below.

Linear conversion example.

This mechanism is also very useful for automatic conversions, for instance to convert feet into meters.

6.5. Configuration example

Here is a job example to interface a commercially available sensor.

Schneider Electric Zelio REG48 Temperature Controller

The REG48 temperature controller family is equipped with a MODBUS interface. You can read the measured temperature in the corresponding input register, specified in the controller programming guide (chapter "Data Address Map"). In this particular case, it is register 41001.

Step 1
Configure the Yocto-RS485 in "MODBUS-RTU", with speed and parity corresponding to the configuration of your REG48 controller.

Step 2
Create a job containing a single periodic task that reads the 41001 input MODBUS register.

Task example to read the temperature on a REG48 MODBUS controller

Step 3
Configure the *genericSensor1* with unit='C and resolution=0.1. The value read in the register is given in tenth of degree, you must therefore configure the mapping to perform a division by 10.

Configuration of the genericSensor function to display the temperature measured by the REG48.

Step 4
Run the job to check that it works by displaying the functions of each module in the VirtualHub main page. If the result is not what you expected, check the module logs.

7. 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-RS485 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.

7.1. Programming paradigm

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

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

Structure of the Yoctopuce API.

The YSensor class

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

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

Programmation

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

From the programming stand point, your Yocto-RS485 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.

7.2. The Yocto-RS485 module

The Yocto-RS485 is an isolated RS485 interface with built-in datalogger.

module : Module

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

serialPort : SerialPort

attribute	type	modifiable ?
logicalName	String	read-only
advertisedValue	String	read-only
rxCount	Integer	read-only
txCount	Integer	read-only
errCount	Integer	read-only
rxMsgCount	Integer	read-only
txMsgCount	Integer	read-only
lastMsg	String	read-only
currentJob	String	read-only
startupJob	String	read-only
jobMaxTask	Integer	read-only
jobMaxSize	Integer	read-only
command	String	read-only
protocol	Type of messaging protocol	read-only
voltageLevel	Enumerated	read-only
serialMode	Serial parameters	read-only

genericSensor1 : GenericSensor
genericSensor2 : GenericSensor
genericSensor3 : GenericSensor
genericSensor4 : GenericSensor
genericSensor5 : GenericSensor
genericSensor6 : GenericSensor
genericSensor7 : GenericSensor
genericSensor8 : GenericSensor
genericSensor9 : GenericSensor

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

unit	String	read-only
currentValue	Fixed-point number	read-only
lowestValue	Fixed-point number	read-only
highestValue	Fixed-point number	read-only
currentRawValue	Fixed-point number	read-only
logFrequency	Frequency	read-only
reportFrequency	Frequency	read-only
advMode	Enumerated	read-only
calibrationParam	Calibration parameters	read-only
resolution	Fixed-point number	read-only
sensorState	Integer	read-only
signalValue	Fixed-point number	read-only
signalUnit	String	read-only
signalRange	Value range	read-only
valueRange	Value range	read-only
signalBias	Fixed-point number	read-only
signalSampling	Enumerated	read-only
enabled	Boolean	read-only

dataLogger : DataLogger

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

files : Files

attribute	type	modifiable ?
logicalName	String	read-only
advertisedValue	String	read-only
filesCount	Integer	read-only
freeSpace	Integer	read-only

7.3. Module

Global parameters control interface for all Yoctopuce devices

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

productName

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

serialNumber

Character string containing the serial number, unique and programmed at the factory. For a Yocto-RS485 module, this serial number always starts with RS485MK1. You can use the serial number to access a given module by software.

logicalName

Character string containing the logical name of the module, initially empty. This attribute can be modified at will by the user. Once initialized to an non-empty value, it can be used to access a given module. If two modules with the same logical name are in the same project, there is no way to

determine which one answers when one tries accessing by logical name. The logical name is limited to 19 characters among `A..Z`,`a..z`,`0..9`,`_`, and `-`.

productId

USB device identifier of the module, preprogrammed to 72 at the factory.

productRelease

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

firmwareRelease

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

persistentSettings

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

luminosity

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

beacon

Activity of the localization beacon of the module.

upTime

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

usbCurrent

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

rebootCountdown

Countdown to use for triggering a reboot of the module.

userVar

32bit integer variable available for user storage.

7.4. SerialPort

serial port control interface, available for instance in the Yocto-RS232, the Yocto-RS485-V2 or the Yocto-Serial

The `YSerialPort` class allows you to fully drive a Yoctopuce serial port. It can be used to send and receive data, and to configure communication parameters (baud rate, bit count, parity, flow control and protocol). Note that Yoctopuce serial ports are not exposed as virtual COM ports. They are meant to be used in the same way as all Yoctopuce devices.

logicalName

Character string containing the logical name of the serial port, initially empty. This attribute can be modified at will by the user. Once initialized to an non-empty value, it can be used to access the serial port directly. If two serial ports 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 serial port, that is automatically advertised up to the parent hub. For a serial port, the advertised value is a hexadecimal signature that changes after each character received. This signature is made of the lower 16 bits of the receive counter, plus the ASCII code of the last character received.

rxCount

Total number of bytes received since last reset.

txCount

Total number of bytes transmitted since last reset.

errCount

Total number of communication errors detected since last reset.

rxMsgCount

Total number of messages received since last reset.

txMsgCount

Total number of messages transmitted since last reset.

lastMsg

Last message fully received (for Line, Frame and Modbus protocols).

currentJob

Name of the job file currently in use.

startupJob

Name of the job file to use when the device is powered on.

jobMaxTask

Maximum number of tasks in a job that the device can handle.

jobMaxSize

Maximum size allowed for job files.

command

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

protocol

Type of protocol used on the serial link.

voltageLevel

Voltage level used on the serial connection.

serialMode

Baud rate, data bits, parity, and stop bits.

7.5. GenericSensor

GenericSensor control interface, available for instance in the Yocto-0-10V-Rx, the Yocto-4-20mA-Rx, the Yocto-RS485-V2 or the Yocto-milliVolt-Rx

The `YGenericSensor` class allows you to read and configure Yoctopuce signal transducers. It inherits from `YSensor` class the core functions to read measurements, to register callback functions, to access the autonomous datalogger. This class adds the ability to configure the automatic conversion between the measured signal and the corresponding engineering unit.

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 an 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 current value of the measure.

unit

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

currentValue

Current value of the measure, in the specified unit, as a floating point number.

lowestValue

Minimal value of the measure, in the specified unit, as a floating point number.

highestValue

Maximal value of the measure, in the specified unit, as a floating point number.

currentRawValue

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

logFrequency

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

reportFrequency

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

advMode

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

calibrationParam

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

resolution

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

sensorState

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

signalValue

Current value of the electrical signal measured 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.

signalBias

Electric signal bias for zero shift adjustment.

signalSampling

Signal sampling method to use.

enabled

Activation state of the input.

7.6. DataLogger

DataLogger control interface, available on most Yoctopuce sensors.

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

logicalName

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

advertisedValue

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

currentRunIndex

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

timeUTC

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

recording

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

autoStart

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

beaconDriven

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

usage

Percentage of datalogger memory in use.

clearHistory

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

7.7. Files

filesystem control interface, available for instance in the Yocto-Buzzer, the Yocto-Color-V2, the Yocto-RS485-V2 or the YoctoHub-Ethernet

The YFiles class is used to access the filesystem embedded on some Yoctopuce devices. This filesystem makes it possible for instance to design a custom web UI (for networked devices) or to add fonts (on display devices).

logicalName

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

filesCount

Number of files currently loaded in the filesystem.

freeSpace

Free space for uploading new files to the filesystem, in bytes.

7.8. What interface: Native, DLL or Service ?

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

Native control

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

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

Native control by DLL

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

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

Control by service

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

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

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

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

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

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

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

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

Support methods for different languages

Limitations of the Yoctopuce libraries

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

7.9. Programming, where to start?

At this point of the user's guide, you should know the main theoretical points of your Yocto-RS485. It is now time to practice. You must download the Yoctopuce library for your favorite programming language from the Yoctopuce web site[2]. Then skip directly to the chapter corresponding to the chosen programming language.

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

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-RS485.

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

8. Using the Yocto-RS485 in command line

When you want to perform a punctual operation on your Yocto-RS485, such as reading a value, assigning a logical name, and so on, you can obviously use the Virtual Hub, but there is a simpler, faster, and more efficient method: the command line API.

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

8.1. Installing

Download the command line API[2]. You do not need to run any setup, simply copy the executables corresponding to your platform/OS in a directory of your choice. You may add this directory to your PATH variable to be able to access these executables from anywhere. You are all set, you only need to connect your Yocto-RS485, open a shell, and start working by typing for example:

```
C:\>YSerialPort any modbusReadInputRegisters 2 1000 1

C:\>YSerialPort any modbusWriteBits 2 16 1
```

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.

8.2. 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.

8.3. Control of the SerialPort function

To control the SerialPort function of your Yocto-RS485, you need the YSerialPort executable file.

For instance, you can launch:

```
C:\>YSerialPort any modbusReadInputRegisters 2 1000 1

C:\>YSerialPort any modbusWriteBits 2 16 1
```

This example uses the "*any*" target to indicate that we want to work on the first SerialPort 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-RS485 module with the *RS485MK1-123456* serial number which you have called "*MyModule*", and its serialPort 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:\>YSerialPort RS485MK1-123456.serialPort describe

C:\>YSerialPort RS485MK1-123456.MyFunction describe

C:\>YSerialPort MyModule.serialPort describe

C:\>YSerialPort MyModule.MyFunction describe

C:\>YSerialPort MyFunction describe
```

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

```
C:\>YSerialPort all describe
```

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

```
C:\>YSerialPort /help
```

8.4. 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 RS485MK1-12346 set_logicalName MonPremierModule

C:\>YModule RS485MK1-12346 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 RS485MK1-12346 set_logicalName MonPremierModule
C:\>YModule RS485MK1-12346 saveToFlash
```

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

```
C:\>YModule -s  RS485MK1-12346 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.

8.5. Limitations

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

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

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

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

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

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

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

9. Using Yocto-RS485 with JavaScript / EcmaScript

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.2. Using Yoctopuce library for JavaScript / EcmaScript 2017

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

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

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

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

Using the official Yoctopuce library for node.js

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

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

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

```
npm install
```

Once done, you can start the example file using:

```
node demo.js
```

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

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

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

Using the Yoctopuce library within a browser (HTML)

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

Using the Yoctoluce library on older JavaScript engines

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

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

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

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

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

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

Backward-compatibility with the old JavaScript library

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

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

by

[3] http://babeljs.io

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

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

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

by

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

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

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

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

you can use:

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

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

You can also rewrite this last asynchronous call as:

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

9.3. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS485. Here is the skeleton of a JavaScript code snipplet to use the SerialPort function.

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

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

// Retrieve the object used to interact with the device
var serialport = YSerialPort.FindSerialPort("RS485MK1-123456.serialPort");

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

```
if(await serialport.isOnline())
{
    // Use serialport.get_serialMode()
    [...]
}
```

Let us look at these lines in more details.

yocto_api and yocto_serialport import

These two import provide access to functions allowing you to manage Yoctopuce modules. `yocto_api` is always needed, `yocto_serialport` is necessary to manage modules containing a serial port, such as Yocto-RS485. Other imports can be useful in other cases, such as `YModule` which can let you enumerate any type of Yoctopuce device.

YAPI.RegisterHub

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

YSerialPort.FindSerialPort

The `FindSerialPort` method allows you to find a serial port 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-RS485 module with serial number *RS485MK1-123456* which you have named "*MyModule*", and for which you have given the *serialPort* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
serialport = YSerialPort.FindSerialPort("RS485MK1-123456.serialPort")
serialport = YSerialPort.FindSerialPort("RS485MK1-123456.MaFonction")
serialport = YSerialPort.FindSerialPort("MonModule.serialPort")
serialport = YSerialPort.FindSerialPort("MonModule.MaFonction")
serialport = YSerialPort.FindSerialPort("MaFonction")
```

`YSerialPort.FindSerialPort` returns an object which you can then use at will to control the serial port.

isOnline

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

modbusWrite* and modbusRead*

The `modbusWrite*()` and `modbusRead*()` method of the object returned by `YFindSerialPort.FindSerialPort` provides MODBUS communication overs the serial port.

A real example, for Node.js

Open a command window (a terminal, a shell...) and go into the directory **example_nodejs/Doc-GettingStarted-Yocto-RS485** within Yoctopuce library for JavaScript / EcmaScript 2017. In there, you will find a file named `demo.js` with the sample code below, which uses the functions explained above, but this time used with all side materials needed to make it work nicely as a small demo.

If your Yocto-RS485 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-RS485 is connected and where you run the VirtualHub.

```
"use strict";

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

```
require('yoctolib-es2017/yocto_serialport.js');

let serialPort;
let target = {slave: 0, reg: 0};
let g_step = 1;
let rl;
async function startDemo() {
    const readline = YAPI._nodeRequire('readline');

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

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

    // Select specified device, or use first available one
    let serial = process.argv[process.argv.length - 1];
    if (serial[8] != '-') {
        // by default use any connected module suitable for the demo
        let anyserial = YSerialPort.FirstSerialPort();
        if (anyserial) {
            let module = await anyserial.module();
            serial = await module.get_serialNumber();
        } else {
            console.log('No matching module connected, check cable !');
            return;
        }
    }
    console.log('Using device ' + serial);
    serialPort = YSerialPort.FirstSerialPort();
    rl = readline.createInterface({
        input: process.stdin,
        output: process.stdout
    });
    console.log('Please enter the MODBUS slave address (1...255)');
    console.log('Slave:');
    rl.on('line', handleInput);
}

async function handleInput(chunk) {
    var val = parseInt(chunk);
    switch (g_step) {
        case 1:
            if (val < 1 || val > 255) {
                console.log("invalid slave number");
            } else {
                target.slave = val;
                g_step++;
                console.log("Slave = " + target.slave);
                console.log("Please select a Coil No (>=1), Input Bit No (>=10001+),");
                console.log("       Register No (>=30001) or Input Register No (>=40001)");
                console.log("No: ");
            }
            break;
        case 2:
            if (val < 1 || val >= 50000 || (val % 10000) == 0) {
                console.log("invalid register number");
            } else {
                target.reg = val;
                await printModbusValue(target.slave, target.reg);
                g_step++;
                console.log("Press ENTER to read again, Q to quit:");
                if ((target.reg % 30000) < 10000) {
                    console.log(" or enter a new value:");
                }
            }
            break;
        case 3:
            if (chunk.charAt(0) == 'q' || chunk.charAt(0) == 'Q') {
                await YAPI.FreeAPI();
                rl.close();
                break;
            }
            if (chunk.charAt(0) != 'r' && chunk.charAt(0) != 'R' && (target.reg % 30000) <
```

```
10000) {
                if (target.reg >= 30001) {
                    await serialPort.modbusWriteRegister(target.slave, target.reg - 30001,
val);
                } else {
                    await serialPort.modbusWriteBit(target.slave, target.reg - 1, val);
                }
            }
            await printModbusValue(target.slave, target.reg);
            console.log("Press R to read again, Q to quit");
            if ((target.reg % 30000) < 10000) {
                console.log(" or enter a new value");
            }
            console.log(": ");
            break;
        default:
            console.log('data: ' + chunk);
    }
}

async function printModbusValue(slave, reg) {
    var val;
    console.log("reg=" + reg + " slave=" + slave);
    if (reg >= 40001) {
        val = (await serialPort.modbusReadRegisters(slave, reg - 40001, 1))[0];
    } else if (reg >= 30001) {
        val = (await serialPort.modbusReadInputRegisters(slave, reg - 30001, 1))[0];
    } else if (reg >= 10001) {
        val = (await serialPort.modbusReadInputBits(slave, reg - 10001, 1))[0];
    } else {
        val = (await serialPort.modbusReadBits(slave, reg - 1, 1))[0];
    }
    console.log("Current value: " + val);
    return val;
}

startDemo();
```

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

```
npm install
```

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

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

Same example, but this time running in a browser

If you want to see how to use the library within a browser rather than with Node.js, switch to the directory **example_html/Doc-GettingStarted-Yocto-RS485**. You will find there a single HTML file, with a JavaScript section similar to the code above, but with a few changes since it has to interact through an HTML page rather than through the JavaScript console.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Hello World</title>
  <script src="../../lib/yocto_api.js"></script>
  <script src="../../lib/yocto_serialport.js"></script>
  <script>
    let serialPort;
    let slave;
    let reg;
    async function startDemo() {

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

```
      // Setup the API to use the VirtualHub on local machine
      let errmsg = new YErrorMsg();
      if (await YAPI.RegisterHub('127.0.0.1', errmsg) != YAPI.SUCCESS) {
        alert('Cannot contact VirtualHub on 127.0.0.1: ' + errmsg.msg);
        return;
      }
      refresh();
    }
    async function refresh() {
      // Select specified device, or use first available one
      let serial = document.getElementById('serial').value;
      if (serial == '') {
        // by default use any connected module suitable for the demo
        let anyserial = YSerialPort.FirstSerialPort();
        if (anyserial) {
          let module = await anyserial.module();
          serial = await module.get_serialNumber();
          document.getElementById('serial').value = serial;
        }
      }
      serialPort = YSerialPort.FirstSerialPort();
      if (await serialPort.isOnline()) {
        // display motor status
        document.getElementById('msg').value = '';
        document.getElementById('main').style.display = '';
      } else {
        document.getElementById('msg').value = 'Module not connected';
      }
      setTimeout(refresh, 500);
    }


    function slavechanged() {
      slave = parseInt(document.getElementById('slaveinput').value);
      if (slave > 0) {
        document.getElementById('slaveinput').style.display = 'none';
        document.getElementById('slavevalue').innerHTML = slave;
        document.getElementById('regspan').style.display = '';
      }
    }


    async function regchanged() {
      reg = parseInt(document.getElementById('reginput').value);
      let res;
      if (reg > 0) {
        let value = await modbus_readvalue(slave, reg);
        document.getElementById('reginput').style.display = 'none';
        document.getElementById('regvalue').innerHTML = reg;
        document.getElementById('valuespan').style.display = '';
        document.getElementById('value').innerHTML = value;
      }
    }

    async function modbus_readvalue(slave, reg) {
      let val;
      if (reg >= 40001) {
        val = (await serialPort.modbusReadRegisters(slave, reg - 40001, 1))[0];
      } else if (reg >= 30001) {
        val = (await serialPort.modbusReadInputRegisters(slave, reg - 30001, 1))[0];
      } else if (reg >= 10001) {
        val = (await serialPort.modbusReadInputBits(slave, reg - 10001, 1))[0];
      } else {
        val = (await serialPort.modbusReadBits(slave, reg - 1, 1))[0];
      }
      console.log("Current value: " + val);
      return val;
    }

    startDemo();
  </script>
</head>
<body>
Module to use: <input id='serial'>
<input id='msg' style='color:red;border:none;' readonly><br>
<span id='main' style='display:none'>
  Please enter the MODBUS slave address (1...255)<br>
  slave:<input id='slaveinput' onchange='javascript:slavechanged()'>
```

```
  <span id='slavevalue'></span><br>
  <span id='regspan' style='display:none'>
  Please select a Coil No (>=1), Input Bit No (>=10001+)<br>
  Input Register No (>=30001) or Register No (>=40001)<br>
  No: <input id='reginput' onchange='javascript:regchanged()'>
  <span id='regvalue'></span><br>
</span>
<span id='valuespan' style='display:none'>
  CurrentValue: <span id='value'></span><br>
</span>
</body>
</html>
```

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

9.4. Control of the module part

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

```
"use strict";

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

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

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

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

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

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

Changing the module settings

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

```
"use strict";

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

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

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

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

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

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

Listing the modules

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

```
"use strict";

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

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

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

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

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

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

9.5. Error handling

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

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

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

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

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

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

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

10. Using Yocto-RS485 with PHP

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

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

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

10.1. Getting ready

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

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

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

10.2. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS485. Here is the skeleton of a PHP code snipplet to use the SerialPort function.

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

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

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

// Retrieve the object used to interact with the device
$serialport = YSerialPort::FindSerialPort("RS485MK1-123456.serialPort");

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

Let's look at these lines in more details.

yocto_api.php and yocto_serialport.php

These two PHP includes provides access to the functions allowing you to manage Yoctopuce modules. `yocto_api.php` must always be included, `yocto_serialport.php` is necessary to manage modules containing a serial port, such as Yocto-RS485.

YAPI::RegisterHub

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

YSerialPort::FindSerialPort

The `YSerialPort::FindSerialPort` function allows you to find a serial port 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-RS485 module with serial number *RS485MK1-123456* which you have named "*MyModule*", and for which you have given the *serialPort* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
$serialport = YSerialPort::FindSerialPort("RS485MK1-123456.serialPort");
$serialport = YSerialPort::FindSerialPort("RS485MK1-123456.MyFunction");
$serialport = YSerialPort::FindSerialPort("MyModule.serialPort");
$serialport = YSerialPort::FindSerialPort("MyModule.MyFunction");
$serialport = YSerialPort::FindSerialPort("MyFunction");
```

`YSerialPort::FindSerialPort` returns an object which you can then use at will to control the serial port.

isOnline

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

modbusWrite* and modbusRead*

The `modbusWrite*()` and `modbusRead*()` method of the object returned by `yFindSerialPort` provides MODBUS communication over the serial port.

A real example

Open your preferred text editor[4], copy the code sample below, save it with the Yoctopuce library files in a location which is accessible to you web server, then use your preferred web browser to access

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

this page. The code is also provided in the directory **Examples/Doc-GettingStarted-Yocto-RS485** of the Yoctopuce library.

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

```
<HTML>
<HEAD>
    <TITLE>Hello World</TITLE>
</HEAD>
<BODY>
<FORM method='get'>
    <?php

    function readModBus($serialPort, $slave, $reg)
    {
        if($reg >= 40001) $res = $serialPort->modbusReadRegisters($slave, $reg - 40001, 1);
        else if($reg >= 30001) $res = $serialPort->modbusReadInputRegisters($slave, $reg -
30001, 1);
        else if($reg >= 10001) $res = $serialPort->modbusReadInputBits($slave, $reg -
10001, 1);
        else $res = $serialPort->modbusReadBits($slave, $reg - 1, 1);
        return $res[0];
    }

    include('yocto_api.php');
    include('yocto_serialport.php');

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

    $address = '127.0.0.1';

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

    $serialPort = YSerialPort::FirstSerialPort();
    if($serialPort == null)
        die("No module found on $address (check USB cable)");

    $slave = "";
    if(isset($_GET["slave"])) $slave = $_GET["slave"];
    print('Please enter the MODBUS slave address (1...255)<br>');
    Print("slave:");
    if($slave == '') {
        Printf("<input name='slave'>");
    } else {
        print("<b>$slave</b><input name='slave' value='$slave' type='hidden'><br>");
        $reg = "";
        if(isset($_GET["reg"])) $reg = $_GET["reg"];
        print("Please select a Coil No (>=1), Input Bit No (>=10001+),<br>");
        print("       Input Register No (>=30001) or Register No (>=40001)<br>");
        Print("No:");
        if($reg == '') Printf("<input name='reg'>");
        else {
            print("<b>$reg</b><input name='reg' value='$reg' type='hidden'><br>");
            $reg = intVal($reg);
            $v = readModBus($serialPort, $slave, $reg);
            print("Current value: <b><span id='value'>$v</span></b><br>");

            if(($reg % 30000) < 10000) {
                printf(" Enter a new value: <input name='value'><br>");
                $value = '';
                if(isset($_GET["value"])) $value = $_GET["value"];
                if($value != '') {
                    if($reg >= 30001)
                        $serialPort->modbusWriteRegister($slave, $reg - 30001,
intval($value));
                    else
                        $serialPort->modbusWriteBit($slave, $reg - 1, intval($value));
                    $v = readModBus($serialPort, $slave, $reg);
                    Print("<script>document.getElementById('value').innerHTML='$v'</
SCRIPT>");
                }
            }
```

```
        }
    }
    YAPI::FreeAPI();
    ?>
    <input type='submit'>
</FORM>
</BODY>
</HTML>
```

10.3. 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
  YAPI::DisableExceptions();

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

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

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

```
?>
<input type='submit' value='refresh'>
</FORM>
</BODY>
</HTML>
```

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

Changing the module settings

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

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

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

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

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

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

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

Listing the modules

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

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

10.4. HTTP callback API and NAT filters

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

The NAT filter: advantages and disadvantages

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

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

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

Responses from request from LAN machines are routed.

But requests from the outside are blocked.

Seeing Internet without being seen provides an enormous security advantage. However, this signifies that you cannot, a priori, set up your own web server at home to control a home automation installation from the outside. A solution to this problem, advised by numerous home automation system dealers, consists in providing outside visibility to your home automation server itself, by

adding a routing rule in the NAT configuration of the DSL router. The issue of this solution is that it exposes the home automation server to external attacks.

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

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

Configuration

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

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

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

Click on the "configure" button on the first line

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

And select "Yocto-API callback".

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

Usage

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

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

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

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

Common issues

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

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

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

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

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

Limitations

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

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

10.5. Error handling

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

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

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

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

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

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

11. Using Yocto-RS485 with C++

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

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

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

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

11.1. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS485. Here is the skeleton of a C++ code snipplet to use the SerialPort function.

```
#include "yocto_api.h"
#include "yocto_serialport.h"

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

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

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

```
YSerialPort *serialport;
serialport = YSerialPort::FindSerialPort("RS485MK1-123456.serialPort");

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

Let's look at these lines in more details.

yocto_api.h et yocto_serialport.h

These two include files provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api.h` must always be used, `yocto_serialport.h` is necessary to manage modules containing a serial port, such as Yocto-RS485.

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.

YSerialPort::FindSerialPort

The `YSerialPort::FindSerialPort` function allows you to find a serial port 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-RS485 module with serial number *RS485MK1-123456* which you have named "*MyModule*", and for which you have given the *serialPort* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
YSerialPort *serialport = YSerialPort::FindSerialPort("RS485MK1-123456.serialPort");
YSerialPort *serialport = YSerialPort::FindSerialPort("RS485MK1-123456.MyFunction");
YSerialPort *serialport = YSerialPort::FindSerialPort("MyModule.serialPort");
YSerialPort *serialport = YSerialPort::FindSerialPort("MyModule.MyFunction");
YSerialPort *serialport = YSerialPort::FindSerialPort("MyFunction");
```

`YSerialPort::FindSerialPort` returns an object which you can then use at will to control the serial port.

isOnline

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

modbusWrite* and modbusRead*

The `modbusWrite*()` and `modbusRead*()` method of the object returned by `yFindSerialPort` provides MODBUS communication over the serial port.

A real example

Launch your C++ environment and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-RS485** 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_serialport.h"
#include <iostream>
#include <stdlib.h>
```

```
using namespace std;

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

  if (YAPI::RegisterHub("usb", errmsg) != YAPI::SUCCESS) {
    cerr << "RegisterHub error : " << errmsg << endl;
    return 1;
  }

  YSerialPort *serialPort;
  if (argc > 1 && string(argv[1]) != "any") {
    serialPort = YSerialPort::FindSerialPort(string(argv[1]));
  } else {
    serialPort = YSerialPort::FirstSerialPort();
    if (serialPort == NULL) {
      cerr << "No module connected (check USB cable)" << endl;
      return 1;
    }
  }
  if (!serialPort->isOnline()) {
    cout << "Module not connected (check identification and USB cable)" << endl;
    return 1;
  }

  int slave, reg, val;
  string cmd;
  do {
    cout << "Please enter the MODBUS slave address (1...255)" << endl;
    cout << "Slave: ";
    cin >> slave;
  } while(slave < 1 || slave > 255);
  do {
    cout << "Please select a Coil No (>=1), Input Bit No (>=10001+)," << endl;
    cout << "       Input Register No (>=30001) or Register No (>=40001)" << endl;
    cout << "No: " ;
    cin >> reg;
  } while(reg < 1 || reg >= 50000 || (reg % 10000) == 0);
  while(serialPort->isOnline()) {
    if(reg >= 40001) {
      val = serialPort->modbusReadRegisters(slave, reg - 40001, 1)[0];
    } else if(reg >= 30001) {
      val = serialPort->modbusReadInputRegisters(slave, reg - 30001, 1)[0];
    } else if(reg >= 10001) {
      val = serialPort->modbusReadInputBits(slave, reg - 10001, 1)[0];
    } else {
      val = serialPort->modbusReadBits(slave, reg - 1, 1)[0];
    }
    cout << "Current value: " << val << endl;
    cout << "Press R to read again, Q to quit";
    if((reg % 30000) < 10000) {
      cout << " or enter a new value";
    }
    cout << ": " << endl;
    cin >> cmd;
    if(cmd == "q" || cmd == "Q") break;
    if (cmd != "r" && cmd != "R" && (reg % 30000) < 10000) {
      val = atoi(cmd.c_str());
      if(reg >= 30001) {
        serialPort->modbusWriteRegister(slave, reg - 30001, val);
      } else {
        serialPort->modbusWriteBit(slave, reg - 1, val);
      }
    }
  }
  YAPI::FreeAPI();
  return 0;
}
```

11.2. Control of the module part

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

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

#include "yocto_api.h"

using namespace std;

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


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

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

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

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

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

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

Changing the module settings

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

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

#include "yocto_api.h"
```

```
using namespace std;

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

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

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

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

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

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

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

Listing the modules

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

```
#include <iostream>

#include "yocto_api.h"

using namespace std;

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

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

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

  YModule *module = YModule::FirstModule();
```

```
  while (module != NULL) {
    cout << module->get_serialNumber() << " ";
    cout << module->get_productName()  << endl;
    module = module->nextModule();
  }
  YAPI::FreeAPI();
  return 0;
}
```

11.3. Error handling

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

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

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

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

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

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

11.4. Integration variants for the C++ Yoctopuce library

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

Integration in source format (recommended)

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

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

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

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

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

Integration as a static library

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

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

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

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

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

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

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

Integration as a dynamic library

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

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

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

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

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

With GCC, the command line to compile is simply:

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

12. Using Yocto-RS485 with Objective-C

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

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

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

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

12.1. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS485. Here is the skeleton of a Objective-C code snipplet to use the SerialPort function.

```
#import "yocto_api.h"
#import "yocto_serialport.h"

...
NSError *error;
[YAPI RegisterHub:@"usb": &error]
...
// On récupère l'objet représentant le module (ici connecté en local sur USB)
serialport = [YSerialPort FindSerialPort:@"RS485MK1-123456.serialPort"];

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

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

```
    // Utiliser [serialport get_serialMode]
    ...
}
```

Let's look at these lines in more details.

yocto_api.h and yocto_serialport.h

These two import files provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api.h` must always be used, `yocto_serialport.h` is necessary to manage modules containing a serial port, such as Yocto-RS485.

[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.

[SerialPort FindSerialPort]

The `[SerialPort FindSerialPort]` function allows you to find a serial port 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-RS485 module with serial number *RS485MK1-123456* which you have named "*MyModule*", and for which you have given the *serialPort* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
YSerialPort *serialport = [SerialPort FindSerialPort:@"RS485MK1-123456.serialPort"];
YSerialPort *serialport = [SerialPort FindSerialPort:@"RS485MK1-123456.MyFunction"];
YSerialPort *serialport = [SerialPort FindSerialPort:@"MyModule.serialPort"];
YSerialPort *serialport = [SerialPort FindSerialPort:@"MyModule.MyFunction"];
YSerialPort *serialport = [SerialPort FindSerialPort:@"MyFunction"];
```

`[SerialPort FindSerialPort]` returns an object which you can then use at will to control the serial port.

isOnline

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

modbusWrite* and modbusRead*

The `modbusWrite*()` and `modbusRead*()` method of the object returned by `YFindSerialPort.FindSerialPort` provides MODBUS communication overs the serial port.

A real example

Launch Xcode 4.2 and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-RS485** of the Yoctopuce library.

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

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

int main(int argc, const char * argv[])
{
  NSError *error;
  char cmd[50] = {0};

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

```
      return 1;
    }

    YSerialPort *serialPort;
    if (argc > 1) {
      NSString     *target = [NSString stringWithUTF8String:argv[1]];
      serialPort = [YSerialPort FindSerialPort:target];
    } else {
      serialPort = [YSerialPort FirstSerialPort];
      if (serialPort == NULL) {
        NSLog(@"No module connected (check USB cable)");
        return 1;
      }
    }

    int slave, reg, val;
    do {
      NSLog(@"Please enter the MODBUS slave address (1...255)");
      NSLog(@"Slave: ");
      fgets(cmd, sizeof(cmd), stdin);
      slave = atoi(cmd);
    } while(slave < 1 || slave > 255);
    do {
      NSLog(@"Please select a Coil No (>=1), Input Bit No (>=10001+),");
      NSLog(@"       Register No (>=30001) or Input Register No (>=40001)");
      NSLog(@"No: ");
      fgets(cmd, sizeof(cmd), stdin);
      reg = atoi(cmd);
    } while(reg < 1 || reg >= 50000 || (reg % 10000) == 0);
    while(true) {
      if(reg >= 40001) {
        val = (int)[[[serialPort modbusReadRegisters:slave :reg - 40001 :1] objectAtIndex:0
]
                    integerValue];
      } else if(reg >= 30001) {
        val = (int)[[[serialPort modbusReadInputRegisters:slave :reg - 30001 :1]
objectAtIndex:0]
                    integerValue];
      } else if(reg >= 10001) {
        val = (int)[[[serialPort modbusReadInputBits:slave :reg - 10001 :1] objectAtIndex:0
]
                    integerValue];
      } else {
        val = (int)[[[serialPort modbusReadBits:slave :reg - 1 :1] objectAtIndex:0]
integerValue];
      }
      NSLog(@"Current value: %d" , val );
      NSLog(@"Press R to read again, Q to quit");
      if((reg % 30000) < 10000) {
        NSLog(@" or enter a new value");
      }
      NSLog(@": ");
      fgets(cmd, sizeof(cmd), stdin);
      if(cmd[0] == 'q' || cmd[0] == 'Q') break;
      if (cmd[0] != 'r' && cmd[0] != 'R' && (reg % 30000) < 10000) {
        val = atoi(cmd);
        if(reg >= 30001) {
          [serialPort modbusWriteRegister:slave :reg - 30001 :val];
        } else {
          [serialPort modbusWriteBit:slave :reg - 1 :val];
        }
      }
    }
    [YAPI FreeAPI];
  }
  return 0;
}
```

12.2. Control of the module part

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

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

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


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

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

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

Changing the module settings

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

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

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

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

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

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

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

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

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

Listing the modules

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

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

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

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

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

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

```
    [YAPI FreeAPI];
  }
  return 0;
}
```

12.3. Error handling

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

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

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

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

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

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

13. Using Yocto-RS485 with Visual Basic .NET

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

13.1. Installation

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

13.2. Using the Yoctopuce API in a Visual Basic project

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

Configuring a Visual Basic project

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

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

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

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

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

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

13.3. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS485. Here is the skeleton of a Visual Basic code snipplet to use the SerialPort function.

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

' Retrieve the object used to interact with the device
Dim serialport As YSerialPort
serialport = YSerialPort.FindSerialPort("RS485MK1-123456.serialPort")

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

[...]
```

Let's look at these lines in more details.

YAPI.RegisterHub

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

YSerialPort.FindSerialPort

The `YSerialPort.FindSerialPort` function allows you to find a serial port 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-RS485 module with serial number *RS485MK1-123456* which you have named "*MyModule*", and for which you have given the *serialPort* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
serialport = YSerialPort.FindSerialPort("RS485MK1-123456.serialPort")
serialport = YSerialPort.FindSerialPort("RS485MK1-123456.MyFunction")
serialport = YSerialPort.FindSerialPort("MyModule.serialPort")
serialport = YSerialPort.FindSerialPort("MyModule.MyFunction")
serialport = YSerialPort.FindSerialPort("MyFunction")
```

`YSerialPort.FindSerialPort` returns an object which you can then use at will to control the serial port.

isOnline

The `isOnline()` method of the object returned by `YSerialPort.FindSerialPort` 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.

modbusWrite* and modbusRead*

The `modbusWrite*()` and `modbusRead*()` method of the object returned by `yFindSerialPort` provides MODBUS communication over the serial port.

A real example

Launch Microsoft VisualBasic and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-RS485** 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.

```
Imports System.IO
Imports System.Environment

Module Module1

  Sub Main()

    Dim argv() As String = System.Environment.GetCommandLineArgs()
    Dim serialPort As YSerialPort
    Dim errmsg = ""
    Dim cmd As String
    Dim slave, reg, val As Integer

    REM Setup the API to use local USB devices. You can
    REM use an IP address instead of 'usb' if the device
    REM is connected to a network.

    If (YAPI.RegisterHub("usb", errmsg) <> YAPI.SUCCESS) Then
      Console.WriteLine("yInitAPI failed: " + errmsg)
      End
    End If

    If (argv.Length > 1 And argv(1) <> "any") Then
      serialPort = YSerialPort.FindSerialPort(argv(1))
    Else
      serialPort = YSerialPort.FirstSerialPort()
      If serialPort Is Nothing Then
        Console.WriteLine("No module connected (check USB cable)")
        End
      End If
    End If

    Console.WriteLine("Please enter the MODBUS slave address (1...255)")
    Console.WriteLine("Slave: ")
    slave = Convert.ToInt32(Console.ReadLine())

    Console.WriteLine("Please select a Coil No (>=1), Input Bit No (>=10001+),")
    Console.WriteLine("       Input Register No (>=30001) or Register No (>=40001)")
    Console.WriteLine("No: ")
    reg = Convert.ToInt32(Console.ReadLine())
    While (serialPort.isOnline())
      If reg >= 40001 Then
        val = serialPort.modbusReadRegisters(slave, reg - 40001, 1)(0)
      ElseIf (reg >= 30001) Then
        val = serialPort.modbusReadInputRegisters(slave, reg - 30001, 1)(0)
      ElseIf (reg >= 10001) Then
        val = serialPort.modbusReadInputBits(slave, reg - 10001, 1)(0)
      Else
        val = serialPort.modbusReadBits(slave, reg - 1, 1)(0)
      End If
      Console.WriteLine("Current value: " + Convert.ToString(val))
      Console.WriteLine("Press ENTER to read again, Q to quit")
      If ((reg Mod 30000) < 10000) Then Console.WriteLine(" or enter a new value")

      cmd = Console.ReadLine()
      If cmd = "q" Or cmd = "Q" Then End

      If (cmd <> "" And (reg Mod 30000) < 10000) Then
        val = Convert.ToInt32(cmd)
        If reg >= 30001 Then
          serialPort.modbusWriteRegister(slave, reg - 30001, val)
        Else
          serialPort.modbusWriteBit(slave, reg - 1, val)
```

```
        End If
      End If
    End While
    YAPI.FreeAPI()
  End Sub

End Module
```

13.4. Control of the module part

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

```
Imports System.IO
Imports System.Environment

Module Module1

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

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

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

    If argv.Length < 2 Then usage()

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

End Module
```

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

Changing the module settings

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

```
Module Module1


  Sub usage()

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

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

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

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

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

  End Sub

End Module
```

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

Listing the modules

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

```
Module Module1

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

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

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

End Module
```

13.5. Error handling

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

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

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

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

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

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

14. Using Yocto-RS485 with C#

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

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

14.1. Installation

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

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

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

Configuring a Visual C# project

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

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

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

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

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

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

14.3. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS485. Here is the skeleton of a C# code snipplet to use the SerialPort function.

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

// Retrieve the object used to interact with the device
YSerialPort serialport = YSerialPort.FindSerialPort("RS485MK1-123456.serialPort");

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

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.

YSerialPort.FindSerialPort

The `YSerialPort.FindSerialPort` function allows you to find a serial port 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-RS485 module with serial number *RS485MK1-123456* which you have named "*MyModule*", and for which you have given the *serialPort* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
serialport = YSerialPort.FindSerialPort("RS485MK1-123456.serialPort");
serialport = YSerialPort.FindSerialPort("RS485MK1-123456.MyFunction");
serialport = YSerialPort.FindSerialPort("MyModule.serialPort");
serialport = YSerialPort.FindSerialPort("MyModule.MyFunction");
serialport = YSerialPort.FindSerialPort("MyFunction");
```

`YSerialPort.FindSerialPort` returns an object which you can then use at will to control the serial port.

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

isOnline

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

modbusWrite* and modbusRead*

The `modbusWrite*()` and `modbusRead*()` method of the object returned by `YFindSerialPort.FindSerialPort` provides MODBUS communication overs the serial port.

A real example

Launch Microsoft Visual C# and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-RS485** 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 Main(string[] args)
    {
      string errmsg = "";

      if (YAPI.RegisterHub("usb", ref errmsg) != YAPI.SUCCESS) {
        Console.WriteLine("RegisterHub error: " + errmsg);
        Environment.Exit(1);
      }

      YSerialPort serialPort;
      if (args.Length > 0 && args[0] != "any") {
        serialPort = YSerialPort.FindSerialPort(args[0]);
      } else {
        serialPort = YSerialPort.FirstSerialPort();
        if (serialPort == null) {
          Console.WriteLine("No module connected (check USB cable)");
          Environment.Exit(1);
        }
      }
      int slave, reg, val;
      String cmd;
      do {
        Console.WriteLine("Please enter the MODBUS slave address (1...255)");
        Console.Write("Slave: ");
        slave = Convert.ToInt32(Console.ReadLine());
      } while (slave < 1 || slave > 255);
      do {
        Console.WriteLine("Please select a Coil No (>=1), Input Bit No (>=10001+),");
        Console.WriteLine("       Input Register No (>=30001) or Register No (>=40001)");
        Console.Write("No: ");
        reg = Convert.ToInt32(Console.ReadLine());
      } while (reg < 1 || reg >= 50000 || (reg % 10000) == 0);
      while (true) {
        if (reg >= 40001) {
          val = serialPort.modbusReadRegisters(slave, reg - 40001, 1)[0];
        } else if (reg >= 30001) {
          val = serialPort.modbusReadInputRegisters(slave, reg - 30001, 1)[0];
        } else if (reg >= 10001) {
          val = serialPort.modbusReadInputBits(slave, reg - 10001, 1)[0];
        } else {
          val = serialPort.modbusReadBits(slave, reg - 1, 1)[0];
        }
        Console.WriteLine("Current value: " + val.ToString());
        Console.Write("Press ENTER to read again, Q to quit");
        if ((reg % 30000) < 10000) {
          Console.Write(" or enter a new value");
        }
```

```
        Console.Write(": ");
        cmd = Console.ReadLine();
        if (cmd == "q" || cmd == "Q") break;
        if (cmd != "" && (reg % 30000) < 10000) {
          val = Convert.ToInt32(cmd);
          if (reg >= 30001) {
            serialPort.modbusWriteRegister(slave, reg - 30001, val);
          } else {
            serialPort.modbusWriteBit(slave, reg - 1, val);
          }
        }
      }
      YAPI.FreeAPI();
    }
  }
}
```

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.

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


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

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

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


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

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

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

        Console.WriteLine("serial:       " + m.get_serialNumber());
        Console.WriteLine("logical name: " + m.get_logicalName());
        Console.WriteLine("luminosity:   " + m.get_luminosity().ToString());
        Console.Write("beacon:       ");
        if (m.get_beacon() == YModule.BEACON_ON)
          Console.WriteLine("ON");
        else
```

```
          Console.WriteLine("OFF");
        Console.WriteLine("upTime:       " + (m.get_upTime() / 1000 ).ToString() + " sec");
        Console.WriteLine("USB current:  " + m.get_usbCurrent().ToString() + " mA");
        Console.WriteLine("Logs:\r\n" + m.get_lastLogs());

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

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

Changing the module settings

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

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

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

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

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

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

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

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

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

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

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

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

Listing the modules

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

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

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

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

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

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

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

15. Using the Yocto-RS485 with Universal Windows Platform

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

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

15.1. Blocking and asynchronous functions

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

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

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

Example:

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

    return result;
}

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

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

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

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

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

15.2. Installation

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

15.3. Using the Yoctopuce API in a Visual Studio project

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

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

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

The Package.appxmanifest file

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

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

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

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

For the Yocto-RS485, here is what you must add in the "Capabilities" node:

```
<DeviceCapability Name="humaninterfacedevice">
    <!-- Yocto-RS485 -->
    <Device Id="vidpid:24e0 0048">
      <Function Type="usage:ff00 0001" />
```

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

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

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

15.4. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS485. Here is the skeleton of a C# code snippet to use the SerialPort function.

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

// Retrieve the object used to interact with the device
YSerialPort serialport = YSerialPort.FindSerialPort("RS485MK1-123456.serialPort");

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

[...]
```

Let us look at these lines in more details.

YAPI.RegisterHub

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

YSerialPort.FindSerialPort

The `YSerialPort.FindSerialPort` function allows you to find a serial port 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-RS485 module with serial number *RS485MK1-123456* which you have named "*MyModule*", and for which you have given the *serialPort* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
serialport = YSerialPort.FindSerialPort("RS485MK1-123456.serialPort");
serialport = YSerialPort.FindSerialPort("RS485MK1-123456.MaFonction");
serialport = YSerialPort.FindSerialPort("MonModule.serialPort");
serialport = YSerialPort.FindSerialPort("MonModule.MaFonction");
serialport = YSerialPort.FindSerialPort("MaFonction");
```

`YSerialPort.FindSerialPort` returns an object which you can then use at will to control the serial port.

isOnline

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

modbusWrite* and modbusRead*

The `modbusWrite*()` and `modbusRead*()` method of the object returned by `YFindSerialPort.FindSerialPort` provides MODBUS communication overs the serial port.

15.5. A real example

Launch Visual Studio and open the corresponding project provided in the directory **Examples/Doc-GettingStarted-Yocto-RS485** of the Yoctopuce library.

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

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

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

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

    public override async Task<int> Run()
    {
      try {
        int slave = Convert.ToInt32(Slave);
        if (slave < 1 || slave > 255) {
          WriteLine("Invalid MODBUS slave address");
          return -1;
        }

        int reg = Convert.ToInt32(Register);
        if (reg < 1 || reg >= 50000 || (reg % 10000) == 0) {
          WriteLine("Invalid MODBUS Register");
          return -1;
        }

        await YAPI.RegisterHub(HubURL);

        YSerialPort serialPort;
        if (Target.ToLower() == "any") {
          serialPort = YSerialPort.FirstSerialPort();
          if (serialPort == null) {
            WriteLine("No module connected (check USB cable) ");
            return -1;
          }
          YModule ymod = await serialPort.get_module();
          WriteLine("Using: " + await ymod.get_serialNumber());
        } else {
          serialPort = YSerialPort.FindSerialPort(Target + ".serialPort");
        }

        int val;
        if (reg >= 40001) {
          val = (await serialPort.modbusReadRegisters(slave, reg - 40001, 1))[0];
        } else if (reg >= 30001) {
          val = (await serialPort.modbusReadInputRegisters(slave, reg - 30001, 1))[0];
        } else if (reg >= 10001) {
          val = (await serialPort.modbusReadInputBits(slave, reg - 10001, 1))[0];
        } else {
          val = (await serialPort.modbusReadBits(slave, reg - 1, 1))[0];
        }

        WriteLine("Current value: " + val.ToString());

        if (Value != "" && (reg % 30000) < 10000) {
```

```
          val = Convert.ToInt32(Value);
          if (reg >= 30001) {
            await serialPort.modbusWriteRegister(slave, reg - 30001, val);
          } else {
            await serialPort.modbusWriteBit(slave, reg - 1, val);
          }
        }
      } catch (YAPI_Exception ex) {
        WriteLine("error: " + ex.Message);
      }

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

15.6. Control of the module part

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

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

namespace Demo
{
  public class Demo : DemoBase
  {

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

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

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

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

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

Changing the module settings

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

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

namespace Demo
{
  public class Demo : DemoBase
  {

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

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

        await YAPI.RegisterHub(HubURL);

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

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

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

Listing the modules

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

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

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

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

        WriteLine("Device list");
        m = YModule.FirstModule();
        while (m != null) {
          WriteLine(await m.get_serialNumber()
                    + " (" + await m.get_productName() + ")");
          m = m.nextModule();
        }
      } catch (YAPI_Exception ex) {
        WriteLine("Error:" + ex.Message);
      }
      YAPI.FreeAPI();
      return 0;
    }
  }
}
```

15.7. Error handling

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

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

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

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

Example:

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

16. Using Yocto-RS485 with Delphi

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

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

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

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

16.1. Preparation

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

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

16.2. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS485. Here is the skeleton of a Delphi code snipplet to use the SerialPort function.

```
uses yocto_api, yocto_serialport;

var errmsg: string;
    serialport: TYSerialPort;

[...]
```

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

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

// Retrieve the object used to interact with the device
serialport = yFindSerialPort("RS485MK1-123456.serialPort")

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

Let's look at these lines in more details.

yocto_api and yocto_serialport

These two units provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api` must always be used, `yocto_serialport` is necessary to manage modules containing a serial port, such as Yocto-RS485.

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.

yFindSerialPort

The `yFindSerialPort` function allows you to find a serial port 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-RS485 module with serial number *RS485MK1-123456* which you have named "*MyModule*", and for which you have given the *serialPort* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
serialport := yFindSerialPort("RS485MK1-123456.serialPort");
serialport := yFindSerialPort("RS485MK1-123456.MyFunction");
serialport := yFindSerialPort("MyModule.serialPort");
serialport := yFindSerialPort("MyModule.MyFunction");
serialport := yFindSerialPort("MyFunction");
```

`yFindSerialPort` returns an object which you can then use at will to control the serial port.

isOnline

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

modbusWrite* and modbusRead*

The `modbusWrite*()` and `modbusRead*()` method of the object returned by `yFindSerialPort` provides MODBUS communication over the serial port.

A real example

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

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

```
program helloworld;
{$APPTYPE CONSOLE}
uses
```

```
  SysUtils,
  Windows,
  yocto_api,
  yocto_serialport;

var
 errmsg,line  : string;
 serialPort : TYserialport;
 slave,reg: integer;
 res : TLongIntArray;
 cmd :string;
 val : integer;
begin

  // Setup the API to use local USB devices. You can
  // use an IP address instead of 'usb' if the device
  // is connected to a network.
  if (YRegisterHub('usb', errmsg) <> YAPI_SUCCESS)  then
    begin
      writeln('RegisterHub error: ' + errmsg);
      halt;
    end;

  if (paramcount>1) then
       serialPort := YFindSerialPort(paramstr(1))
    else
     begin
       serialPort := YFirstSerialPort();
       if  (serialPort=nil) then
         begin
           writeln('No module connected (check cable)');
           halt;
         end;
     end;

  writeln('Please enter the MODBUS slave address (1...255)');
  repeat
   ReadLn(slave);
  until (slave>0) and (slave<256);

  writeln('Please select a Coil No (>=1), Input Bit No (>=10001+),');
  writeln('Input Register No (>=30001) or Register No (>=40001)');
  writeln('No: ');
  repeat
  ReadLn(reg);
  until (reg >=1) and  (reg<50000) and ((reg mod 10000)<> 0);

  while (true)  do
   begin
    if (reg>=40001) then res := serialPort.modbusReadRegisters(slave, reg-40001, 1)
    else if (reg>=30001) then res := serialPort.modbusReadInputRegisters(slave, reg-30001,
1)
    else if (reg>=10001) then res := serialPort.modbusReadInputBits(slave, reg-10001, 1)
    else res := serialPort.modbusReadBits(slave, reg-1, 1);
    val := res[0];
    writeln('Current value: '+inttostr(val));
    write('Press ENTER to read again, Q to quit');
    if((reg mod 30000) < 10000) then write (' or enter a new value');
    write(': ');
    readLn(cmd);
    if (cmd ='q') or  (cmd ='Q') then halt;
    if  (cmd<>'') and ((reg mod 30000) < 10000) then
     begin
         val := strtoint(cmd);
         if(reg >= 30001) then serialPort.modbusWriteRegister(slave, reg-30001, val)
                          else    serialPort.modbusWriteBit(slave, reg-1, val);
     end;
   end;
  yFreeAPI();

end.
```

16.3. Control of the module part

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

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

const
  serial = 'RS485MK1-123456'; // use serial number or logical name

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


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

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

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

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

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

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

Changing the module settings

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

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

const
  serial = 'RS485MK1-123456'; // use serial number or logical name

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

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

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

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

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

Listing the modules

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

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

```
var
  module : TYModule;
  errmsg : string;

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

  Writeln('Device list');

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

end.
```

16.4. Error handling

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

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

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

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

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

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

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

17. Using the Yocto-RS485 with Python

Python is an interpreted object oriented language developed by Guido van Rossum. Among its advantages is the fact that it is free, and the fact that it is available for most platforms, Windows as well as UNIX. It is an ideal language to write small scripts on a napkin. The Yoctopuce library is compatible with Python 2.6+ and 3+. It works under Windows, Mac OS X, and Linux, Intel as well as ARM. The library was tested with Python 2.6 and Python 3.2. Python interpreters are available on the Python web site[1].

17.1. Source files

The Yoctopuce library classes[2] for Python that you will use are provided as source files. Copy all the content of the *Sources* directory in the directory of your choice and add this directory to the *PYTHONPATH* environment variable. If you use an IDE to program in Python, refer to its documentation to configure it so that it automatically finds the API source files.

17.2. 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.

17.3. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS485. Here is the skeleton of a Python code snipplet to use the SerialPort function.

[1] http://www.python.org/download/
[2] www.yoctopuce.com/EN/libraries.php

```
[...]
# Enable detection of USB devices
errmsg=YRefParam()
YAPI.RegisterHub("usb",errmsg)
[...]

# Retrieve the object used to interact with the device
serialport = YSerialPort.FindSerialPort("RS485MK1-123456.serialPort")

# Hot-plug is easy: just check that the device is online
if serialport.isOnline():
    # Use serialport.get_serialMode()
    [...]

[...]
```

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.

YSerialPort.FindSerialPort

The `YSerialPort.FindSerialPort` function allows you to find a serial port 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-RS485 module with serial number *RS485MK1-123456* which you have named "*MyModule*", and for which you have given the *serialPort* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
serialport = YSerialPort.FindSerialPort("RS485MK1-123456.serialPort")
serialport = YSerialPort.FindSerialPort("RS485MK1-123456.MyFunction")
serialport = YSerialPort.FindSerialPort("MyModule.serialPort")
serialport = YSerialPort.FindSerialPort("MyModule.MyFunction")
serialport = YSerialPort.FindSerialPort("MyFunction")
```

`YSerialPort.FindSerialPort` returns an object which you can then use at will to control the serial port.

isOnline

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

modbusWrite* and modbusRead*

The `modbusWrite*()` and `modbusRead*()` method of the object returned by `YFindSerialPort.FindSerialPort` provides MODBUS communication overs the serial port.

A real example

Launch Python and open the corresponding sample script provided in the directory **Examples/Doc-GettingStarted-Yocto-RS485** 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_serialport import *
```

```
# Setup the API to use local USB devices. You can
# use an IP address instead of 'usb' if the device
# is connected to a network.

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

if len(sys.argv) > 1:
    serialPort = YSerialPort.FindSerialPort(sys.argv[1])
else:
    serialPort = YSerialPort.FirstSerialPort()
    if serialPort is None:
        sys.exit('No module connected (check cable)')

print("Please enter the MODBUS slave address (1...255)")
slave = 0
while (slave < 1) or (slave > 255):
    slave = int(input("slave: "))  # use raw_input in python 2.x

reg = 0
while (reg < 1) or (reg >= 50000) or (reg % 10000) == 0:
    print("Please select a Coil No (>=1), Input Bit No (>=10001+),")
    print("Input Register No (>=30001) or Register No (>=40001)")
    reg = int(input("No: "))  # use raw_input in python 2.x

while serialPort.isOnline():
    if reg >= 40001:
        val = serialPort.modbusReadRegisters(slave, reg - 40001, 1)[0]
    elif reg >= 30001:
        val = serialPort.modbusReadInputRegisters(slave, reg - 30001, 1)[0]
    elif reg >= 10001:
        val = serialPort.modbusReadInputBits(slave, reg - 10001, 1)[0]
    else:
        val = serialPort.modbusReadBits(slave, reg - 1, 1)[0]

    print("Current value: " + str(val))
    print("Press ENTER to read again, Q to quit")
    if (reg % 30000) < 10000:
        print(" or enter a new value")

    cmd = input(": ")  # use raw_input in python 2.x
    if (cmd == "q") or (cmd == "Q"):
        sys.exit()

    if cmd != "" and ((reg % 30000) < 10000):
        val = int(cmd)
        if reg >= 30001:
            serialPort.modbusWriteRegister(slave, reg - 30001, val)
        else:
            serialPort.modbusWriteBit(slave, reg - 1, val)
YAPI.FreeAPI()
```

17.4. Control of the module part

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

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

from yocto_api import *


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


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

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

```
    usage()

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

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

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

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

Changing the module settings

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

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

from yocto_api import *


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


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

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

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

Warning: the number of write cycles of the nonvolatile memory of the module is limited. When this limit is reached, nothing guaranties that the saving process is performed correctly. This limit, linked to the technology employed by the module micro-processor, is located at about 100000 cycles. In short,

you can use the `YModule.saveToFlash()` function only 100000 times in the life of the module. Make sure you do not call this function within a loop.

Listing the modules

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

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


from yocto_api import *

errmsg = YRefParam()

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

print('Device list')

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

17.5. Error handling

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

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

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

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

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

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

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

18. Using the Yocto-RS485 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.

18.1. Getting ready

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

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

The library is available as source files as well as a *jar* file. Decompress the library files in a folder of your choice, connect your modules, run the VirtualHub software, and you are ready to start your first tests. You do not need to install any driver.

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

18.2. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS485. Here is the skeleton of a Java code snippet to use the SerialPort function.

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

// Retrieve the object used to interact with the device
serialport = YSerialPort.FindSerialPort("RS485MK1-123456.serialPort");

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

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

```
    // Use serialport.get_serialMode()
    [...]
}

[...]
```

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.

YSerialPort.FindSerialPort

The `YSerialPort.FindSerialPort` function allows you to find a serial port 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-RS485 module with serial number *RS485MK1-123456* which you have named "*MyModule*", and for which you have given the *serialPort* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
serialport = YSerialPort.FindSerialPort("RS485MK1-123456.serialPort")
serialport = YSerialPort.FindSerialPort("RS485MK1-123456.MyFunction")
serialport = YSerialPort.FindSerialPort("MyModule.serialPort")
serialport = YSerialPort.FindSerialPort("MyModule.MyFunction")
serialport = YSerialPort.FindSerialPort("MyFunction")
```

`YSerialPort.FindSerialPort` returns an object which you can then use at will to control the serial port.

isOnline

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

modbusWrite* and modbusRead*

The `modbusWrite*()` and `modbusRead*()` method of the object returned by `YFindSerialPort.FindSerialPort` provides MODBUS communication overs the serial port.

A real example

Launch you Java environment and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-RS485** 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.*;
import java.io.BufferedReader;
import java.io.InputStreamReader;

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);
        }

        YSerialPort serialPort;
        if (args.length > 0) {
```

```
            serialPort = YSerialPort.FindSerialPort(args[0]);
        } else {
            serialPort = YSerialPort.FirstSerialPort();
            if (serialPort == null) {
                System.out.println("No module connected (check USB cable)");
                System.exit(1);
            }
        }

        int slave, reg, val;
        String cmd;

        InputStreamReader inputStreamReader = new InputStreamReader(System.in);
        BufferedReader console = new BufferedReader(inputStreamReader);
        try {
            do {
                System.out.println("Please enter the MODBUS slave address (1...255)");
                System.out.print("Slave: ");
                slave = Integer.parseInt(console.readLine());
            } while(slave < 1 || slave > 255);
            do {
                System.out.println("Please select a Coil No (>=1), Input Bit No
(>=10001+),");
                System.out.println("       Input Register No (>=30001) or Register No
(>=40001)");
                System.out.print("No: ");
                reg = Integer.parseInt(console.readLine());
            } while(reg < 1 || reg >= 50000 || (reg % 10000) == 0);
            while(true) {
                if(reg >= 40001) {
                    val = serialPort.modbusReadRegisters(slave, reg-40001, 1).get(0);
                } else if(reg >= 30001) {
                    val = serialPort.modbusReadInputRegisters(slave, reg-30001, 1).get(0);
                } else if(reg >= 10001) {
                    val = serialPort.modbusReadInputBits(slave, reg-10001, 1).get(0);
                } else {
                    val = serialPort.modbusReadBits(slave, reg-1, 1).get(0);
                }
                System.out.println("Current value: "+Integer.toString(val));
                System.out.print("Press ENTER to read again, Q to quit");
                if((reg % 30000) < 10000) {
                    System.out.print(" or enter a new value");
                }
                System.out.print(": ");
                cmd = console.readLine();
                if(cmd.equals("q") || cmd.equals("Q")) break;
                if(!cmd.equals("") && (reg % 30000) < 10000) {
                    val = Integer.parseInt(cmd);
                    if(reg >= 30001) {
                        serialPort.modbusWriteRegister(slave, reg-30001, val);
                    } else {
                        serialPort.modbusWriteBit(slave, reg-1, val);
                    }
                }
            }
        } catch(Exception ex) {
            ex.printStackTrace();
        }

        YAPI.FreeAPI();
    }
}
```

18.3. 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();
    }
}
```

18.4. 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.

19. Using the Yocto-RS485 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.

19.1. 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.

19.2. Getting ready

Go to the Yoctopuce web site and download the Java for Android programming library[1]. The library is available as source files, and also as a jar file. Connect your modules, decompress the library files in the directory of your choice, and configure your Android programming environment so that it can find them.

To keep them simple, all the examples provided in this documentation are snippets of Android applications. You must integrate them in your own Android applications to make them work. However, your can find complete applications in the examples provided with the Java for Android library.

19.3. 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].

19.4. Activating the USB port under Android

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

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

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

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

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

it stays connected. To enable the Yoctopuce library to correctly manage these authorizations, your must provide a pointer on the application context by calling the EnableUSBHost method of the YAPI class before the first USB access. This function takes as arguments an object of the `android.content.Context` class (or of a subclass). As the Activity class is a subclass of `Context`, it is simpler to call `YAPI.EnableUSBHost(this);` in the method `onCreate` of your application. If the object passed as parameter is not of the correct type, a `YAPI_Exception` exception is generated.

```
...
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    try {
                // Pass the application Context to the Yoctopuce Library
        YAPI.EnableUSBHost(this);
        } catch (YAPI_Exception e) {
                Log.e("Yocto",e.getLocalizedMessage());
        }
}
...
```

Autorun

It is possible to register your application as a default application for a USB module. In this case, as soon as a module is connected to the system, the application is automatically launched. You must add *<action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"/>* in the section *<intent-filter>* of the main activity. The section *<activity>* must have a pointer to an XML file containing the list of USB modules which can run the application.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    ...
    <uses-feature android:name="android.hardware.usb.host" />
    ...
    <application ... >
        <activity
            android:name=".MainActivity" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>

            <meta-data
                android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"
                android:resource="@xml/device_filter" />
        </activity>
    </application>

</manifest>
```

The XML file containing the list of modules allowed to run the application must be saved in the res/xml directory. This file contains a list of USB *vendorId* and *deviceID* in decimal. The following example runs the application as soon as a Yocto-Relay or a YoctoPowerRelay is connected. You can find the vendorID and the deviceID of Yoctopuce modules in the characteristics section of the documentation.

```
<?xml version="1.0" encoding="utf-8"?>

<resources>
    <usb-device vendor-id="9440" product-id="12" />
    <usb-device vendor-id="9440" product-id="13" />
</resources>
```

19.5. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS485. Here is the skeleton of a Java code snippet to use the SerialPort function.

```
[...]
// Enable detection of USB devices
YAPI.EnableUSBHost(this);
YAPI.RegisterHub("usb");
[...]
// Retrieve the object used to interact with the device
serialport = YSerialPort.FindSerialPort("RS485MK1-123456.serialPort");

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

[...]
```

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.

YSerialPort.FindSerialPort

The `YSerialPort.FindSerialPort` function allows you to find a serial port 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-RS485 module with serial number *RS485MK1-123456* which you have named "*MyModule*", and for which you have given the *serialPort* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
serialport = YSerialPort.FindSerialPort("RS485MK1-123456.serialPort")
serialport = YSerialPort.FindSerialPort("RS485MK1-123456.MyFunction")
serialport = YSerialPort.FindSerialPort("MyModule.serialPort")
serialport = YSerialPort.FindSerialPort("MyModule.MyFunction")
serialport = YSerialPort.FindSerialPort("MyFunction")
```

`YSerialPort.FindSerialPort` returns an object which you can then use at will to control the serial port.

isOnline

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

modbusWrite* and modbusRead*

The `modbusWrite*()` and `modbusRead*()` method of the object returned by `YFindSerialPort.FindSerialPort` provides MODBUS communication overs the serial port.

A real example

Launch you Java environment and open the corresponding sample project provided in the directory **Examples//Doc-Examples** of the Yoctopuce library.

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

```
package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemSelectedListener;
import android.widget.ArrayAdapter;
import android.widget.EditText;
import android.widget.Spinner;
import android.widget.Switch;
import android.widget.TextView;
import android.widget.Toast;

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

public class GettingStarted_Yocto_RS485 extends Activity implements OnItemSelectedListener
{

    private ArrayAdapter<String> aa;
    private YModule module = null;
    private TextView resultTextView;
    private EditText valueEditText;
    private EditText registerEditText;
    private EditText slaveEditText;
    private Spinner my_spin;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.gettingstarted_yocto_rs485);
        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);
        slaveEditText = (EditText) findViewById(R.id.slavefield);
        registerEditText = (EditText) findViewById(R.id.registerfield);
        valueEditText = (EditText) findViewById(R.id.valuefield);
        resultTextView = (TextView) findViewById(R.id.resultvalue);
    }

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

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

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

```
    }

    private int _doModbus(String hwid, String slavefield, String registerfield, String
cmdfield)
    {
        int slave;
        int reg;
        try {
            slave = Integer.parseInt(slavefield);
            reg = Integer.parseInt(registerfield);
        } catch (NumberFormatException ex) {
            Toast.makeText(this,ex.toString(),Toast.LENGTH_LONG).show();
            return 0;
        }
        try {
            YSerialPort serialPort = YSerialPort.FindSerialPort(hwid);
            // send new value to modbus device
            if(!cmdfield.equals("") && (reg % 30000) < 10000) {
                int cmd = Integer.parseInt(cmdfield);
                if(reg >= 30001) {
                    serialPort.modbusWriteRegister(slave, reg-30001, cmd);
                } else {
                    serialPort.modbusWriteBit(slave, reg-1, cmd);
                }
            }
            // read it again

            int val;
            if(reg >= 40001) {
                val = serialPort.modbusReadRegisters(slave, reg-40001, 1).get(0);
            } else if(reg >= 30001) {
                val = serialPort.modbusReadInputRegisters(slave, reg-30001, 1).get(0);
            } else if(reg >= 10001) {
                val = serialPort.modbusReadInputBits(slave, reg-10001, 1).get(0);
            } else {
                val = serialPort.modbusReadBits(slave, reg-1, 1).get(0);
            }
            return val;
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
        return 0;
    }

    @Override
    public void onItemSelected(AdapterView<?> parent, View view, int pos, long id)
    {
        resultTextView.setText("");
    }

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

    public void refreshInfo(View view)
    {
        Object selectedItem = my_spin.getSelectedItem();
        if (selectedItem!=null) {
            String hwid = selectedItem.toString();
            int val = _doModbus(hwid, slaveEditText.getText().toString(),
                    registerEditText.getText().toString(), valueEditText.getText().toString
());
            resultTextView.setText(Integer.toString(val));
        }
    }


}
```

19.6. Control of the module part

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

```
package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemSelectedListener;
import android.widget.ArrayAdapter;
import android.widget.Spinner;
import android.widget.Switch;
import android.widget.TextView;

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

public class ModuleControl extends Activity implements OnItemSelectedListener
{

    private ArrayAdapter<String> aa;
    private YModule module = null;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.modulecontrol);
        Spinner my_spin = (Spinner) findViewById(R.id.spinner1);
        my_spin.setOnItemSelectedListener(this);
        aa = new ArrayAdapter<String>(this, android.R.layout.simple_spinner_item);
        aa.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
        my_spin.setAdapter(aa);
    }

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

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

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

    private void DisplayModuleInfo()
    {
        TextView field;
        if (module == null)
            return;
        try {
            field = (TextView) findViewById(R.id.serialfield);
            field.setText(module.getSerialNumber());
            field = (TextView) findViewById(R.id.logicalnamefield);
            field.setText(module.getLogicalName());
            field = (TextView) findViewById(R.id.luminosityfield);
            field.setText(String.format("%d%%", module.getLuminosity()));
            field = (TextView) findViewById(R.id.uptimefield);
            field.setText(module.getUpTime() / 1000 + " sec");
            field = (TextView) findViewById(R.id.usbcurrentfield);
```

```
            field.setText(module.getUsbCurrent() + " mA");
            Switch sw = (Switch) findViewById(R.id.beaconswitch);
            sw.setChecked(module.getBeacon() == YModule.BEACON_ON);
            field = (TextView) findViewById(R.id.logs);
            field.setText(module.get_lastLogs());

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

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

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

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

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

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

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

Changing the module settings

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

```
package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemSelectedListener;
import android.widget.ArrayAdapter;
import android.widget.EditText;
import android.widget.Spinner;
import android.widget.TextView;
import android.widget.Toast;
```

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

public class SaveSettings extends Activity implements OnItemSelectedListener
{

    private ArrayAdapter<String> aa;
    private YModule module = null;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.savesettings);
        Spinner my_spin = (Spinner) findViewById(R.id.spinner1);
        my_spin.setOnItemSelectedListener(this);
        aa = new ArrayAdapter<String>(this, android.R.layout.simple_spinner_item);
        aa.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
        my_spin.setAdapter(aa);
    }

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

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

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

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

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

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

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

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

}
```

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

Listing the modules

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

```
package com.yoctopuce.doc_examples;

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

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

public class Inventory extends Activity
{

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

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

        try {
            YAPI.UpdateDeviceList();
            YModule module = YModule.FirstModule();
            while (module != null) {
                String line = module.get_serialNumber() + " (" + module.get_productName() +
")";
                TextView tx = new TextView(this);
                tx.setText(line);
                tx.setTextSize(TypedValue.COMPLEX_UNIT_SP, 20);
```

```
                layout.addView(tx);
                module = module.nextModule();
            }
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
    }

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

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

}
```

19.7. 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.

20. Using the Yocto-RS485 with LabVIEW

LabVIEW is edited by National Instruments since 1986. It is a graphic development environment: rather than writing lines of code, the users draw their programs, somewhat like a flow chart. LabVIEW was designed mostly to interface measuring tools, hence the *Virtual Instruments* name for LabVIEW programs. With visual programming, drawing complex algorithms becomes quickly fastidious. The LabVIEW Yoctopuce library was thus designed to make it as easy to use as possible. In other words, LabVIEW being an environment extremely different from other languages supported by Yoctopuce, there are major differences between the LabVIEW API and the other APIs.

20.1. Architecture

The LabVIEW library is based on the Yoctopuce DotNetProxy library contained in the DotNetProxyLibrary.dll DLL. In fact, it is this DotNetProxy library which takes care or most of the work by relying on the C# library which, in turn, uses the low level library coded in yapi.dll (32bits) and amd64\yapi.dll(64bits).

LabVIEW Yoctopuce API architecture

You must therefore imperatively distribute the *DotNetProxyLibrary.dll*, *yapi.dll*, and *amd64\yapi.dll* with your LabVIEW applications using the Yoctopuce API.

If need be, you can find the low level API sources in the C# library and the *DotNetProxyLibrary.dll* sources in the *DotNetProxy* library.

20.2. Compatibility

Firmware

For the LabVIEW Yoctopuce library to work correctly with your Yoctopuce modules, these modules need to have firmware 37120, or higher.

LabVIEW for Linux and MacOS

At the time of writing, the LabVIEW Yoctopuce API has been tested under Windows only. It is therefore most likely that it simply does not work with the Linux and MacOS versions of LabVIEW.

LabVIEW NXG

The LabVIEW Yoctopuce library uses many techniques which are not yet available in the new generation of LabVIEW. The library is therefore absolutely not compatible with LabVIEW NXG.

About DotNewProxyLibrary.dll

In order to be compatible with as many versions of Windows as possible, including Windows XP, the *DotNetProxyLibrary.dll* library is compiled in .NET 3.5, which is available by default on all the Windows versions since XP.

20.3. Installation

Download the LabVIEW library from the Yoctopuce web site[1]. It is a ZIP file in which there is a distinct directory for each version of LabVIEW. Each of these directories contains two subdirectories: the first one contains programming examples for each Yoctopuce product; the second one, called *VIs*, contains all the VIs of the API and the required DLLs.

Depending on Windows configuration and the method used to copy the *DotNetProxyLibrary.dll* on your system, Windows may block it because it comes from an other computer. This may happen when the library zip file is uncompressed with Window's file explorer. If the DLL is blocked, LabVIEW will not be able to load it and an error 1386 will occur whenever any of the Yoctopuce VIs is executed.

There are two ways to fix this. The simplest is to unblock the file with the Windows file explorer: *right click / properties* on the *DotNetProxyLibrary.dll* file, and click on the *unblock* button. But this has to be done each time a new version of the DLL is copied on your system.

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

Unblock the DotNetProxyLibrary DLL.

Alternatively, one can modify the LabVIEW configuration by creating, in the same directory as the labview.exe executable, an XML file called *labview.exe.config* containing the following code:

```
<?xml version ="1.0"?>
<configuration>
 <runtime>
 <loadFromRemoteSources enabled="true" />
 </runtime>
</configuration>
```

Make sure to select the correct directory depending on the LabVIEW version you are using (32 bits vs. 64 bits). You can find more information about this file on the National Instruments web site.[2]

To install the LabVIEW Yoctopuce API, there are several methods.

Method 1 : "Take-out" installation

The simplest way to use the Yoctopuce library is to copy the content of the *VIs* directory wherever you want and to use the VIs in LabVIEW with a simple drag-n-drop operation.

To use the examples provided with the API, it is simpler if you add the directory of Yoctopuce VIs into the list of where LabVIEW must look for VIs that it has not found. You can access this list through the *Tools > Options > Paths > VI Search Path* menu.

[2] https://knowledge.ni.com/KnowledgeArticleDetails?id=kA00Z000000P8XnSAK

Configuring the "VI Search Path"

Method 2 : Provided installer

In each LabVIEW folder of the Library, you will find a VI named "*Install.vi*", just open the one matching your LabVIEW version.

The provider installer

This installer provide 3 installation options:

Install: Keep VI and documentation files where they are.
With this option, VI files are keep in the place where the library has been unzipped. So you will have to make sure these files are not deleted as long as you need them. Here is what the installer will do if that option is chosen:

- All references to Yoctopuce any library paths will be removed from the *viSearchPath* option in the *labview.ini* file.
- A dir.mnu palette file referring to VIs in the install folder will be created in *C:\Program Files xx\National Instruments\LabVIEW 20xx\vi.lib\addons\Yoctopuce*
- A reference to the VIs source install path will inserted into the *viSearchPath* option in the *labview.ini* file.

Install: Copy VI and documentation files into LabVIEW's vi.lib folder
In that case all required files are copied inside the LabVIEW's installation folder, so you will be able to delete the installation folder once the original installation is complete. Note that programming examples won't be copied. Here is the exact behaviour of the installer in that case:

- All references to Yoctopuce library paths will be removed from *viSearchPath* in *labview.ini* file
- All VIs, DLLs, and documentation files will be copied into:
  C:\Program Files xx\National Instruments\LabVIEW 20xx\vi.lib\Yoctopuce
- VIs will be patched with the path to copied documentation files
- A dir.mnu palette file referring to copied VIs will be created in
  C:\Program Files xx\National Instruments\LabVIEW 20xx\vi.lib\addons\Yoctopuce

Uninstall Yoctopuce Library
this option is meant to remove the LabVIEW library from your LabVIEW installation, here is how it is done:

- All references to Yoctopuce library paths will be removed from *viSearchPath* in *labview.ini* file
- Following folders, if exists, will be removed:
  C:\Program Files xx\National Instruments\LabVIEW 20xx\vi.lib\addons\Yoctopuce
  C:\Program Files xx\National Instruments\LabVIEW 20xx\vi.lib\Yoctopuce

In any case, if the *labview.ini* file needs to be modified, a backup copy will be made beforehand.

The installer identifies Yoctopuce VIs library folders by checking the presence of the YRegisterHub.vi file in said folders.

Once the installation is complete, a Yoctopuce palette will appear in *Functions/Addons* menu.

Method 3 : Installation in a LabVIEW palette (ancillary method)

The steps to manually install the VIs directly in the LabVIEW palette are somewhat more complex. You can find the detailed procedure on the National Instruments web site [3], but here is a summary:

1. Create a *Yoctopuce/API* directory in the *C:\Program Files\National Instruments\LabVIEW xxxx \vi.lib* directory and copy all the VIs and DLLs of the *VIs* directory into it.
2. Create a *Yoctopuce* directory in the *C:\Program Files\National Instruments\LabVIEW xxxx \menus\Categories* directory.
3. Run LabVIEW and select the option *Tools>Advanced>Edit Palette Set*

[3] https://forums.ni.com/t5/Developer-Center-Resources/Creating-a-LabVIEW-Palette/ta-p/3520557

Three windows pop up:

- "Edit Controls and Functions Palette Set"
- "Functions"
- "Controls"

.
In the *Function* window, there is a *Yoctopuce* icon. Double-click it to create an empty "Yoctopuce" window.

4. In the Yoctopuce window, perform a *Right click>Insert>Vi(s)..*

in order to open a file chooser. Put the file chooser in the *vi.lib\Yoctopuce\API* directory that you have created in step 1 and click on *Current Folder*

All the Yoctopuce VIs now appear in the Yoctopuce window. By default, they are sorted by alphabetical order, but you can arrange them as you see fit by moving them around with the mouse. For the palette to be easy to use, we recommend to reorganize the icons over 8 columns.

5. In the "*Edit Controls and Functions Palette Set*" window, click on the "Save Changes" button, the window indicates that it has created a `dir.mnu` file in your `Documents` directory.

Copy this file in the "`menus\Categories\Yoctopuce`" directory that you have created in step 2.

6. Restart LabVIEW, the LabVIEW palette now contains a Yoctopuce sub-palette with all the VIs of the API.

20.4. Presentation of Yoctopuce VIs

The LabVIEW Yoctopuce library contains one VI per class of the Yoctopuce API, as well as a few special VIs. All the VIs have the traditional connectors *Error IN* and *Error Out*.

YRegisterHub

The `YRegisterHub` VI is used to initialize the API. You must imperatively call this VI once before you do anything in relation with Yoctopuce modules.

The YRegisterHub VI

The `YRegisterHub` VI takes a *url* parameter which can be:

- The `"usb"` character string to indicated that you wish to work with local modules, directly connected by USB
- An IP address to indicate that you wish to work with modules which are available through a network connection. This IP address can be that of a YoctoHub[4] or even that of a machine on which the VirtualHub[5] application is running.

In the case of an IP address, the `YRegisterHub` VI tries to contact this address and generates and error if it does not succeed, unless the *async* parameter is set to TRUE. If *async* is set to TRUE, no error is generated and Yoctopuce modules corresponding to that IP address become automatically available as soon as the said machine can be reached.

If everything went well, the *successful* output contains the value TRUE. In the opposite case, it contains the value FALSE and the *error msg* output contains a string of characters with a description of the error.

You can use several `YRegisterHub` VIs with distinct URLs if you so wish. However, on the same machine, there can be only one process accessing local Yoctopuce modules directly by USB (*url* set to `"usb"`). You can easily work around this limitation by running the VirtualHub software on the local machine and using the `"127.0.0.1"` url.

YFreeAPI

The `YFreeAPI` VI enables you to free resources allocated by the Yoctopuce API.

The YFreeAPI VI

You must call the `YFreeAPI` VI when your code is done with the Yoctopuce API. Otherwise, direct USB access (*url* set to `"usb"`) could stay locked after the execution of your VI, and stay so for as long as LabVIEW is not completely closed.

Structure of the VIs corresponding to a class

The other VIs correspond to each function/class of the Yoctopuce API, they all have the same structure:

Structure of most VIs of the API.

- Connector [11]: *name* must contain the hardware name or the logical name of the intended function.
- Connectors [10] and [9]: input parameters depending on the nature of the VI.
- Connectors [8] and [0] : *error in* and *error out*.
- Connector [7] : Unique hardware name of the found function.
- Connector [5] : *is online* contains TRUE if the function is available, FALSE otherwise.
- Connectors [2] and [1]: output values depending on the nature of the VI.
- Connector [6]: If this input is set to TRUE, connector [3] contains a reference to the *Proxy* objects implemented by the VI[6]. This input is initialized to FALSE by default.

[4] www.yoctopuce.com/EN/products/category/extensions-and-networking
[5] http://www.yoctopuce.com/EN/virtualhub.php
[6] see section *Using Proxy objects*

- Connector [3]: Reference on the *Proxy* object implemented by the VI if input [6] is TRUE. This object enables you to access additional features.

You can find the list of functions available on your Yocto-RS485 in chapter *Programming, general concepts*.

If the desired function (parameter *name*) is not available, this does not generate an error, but the *is online* output contains FALSE and all the other outputs contain the value "N/A" whenever possible. If the desired function becomes available later in the life of your program, *is online* switches to TRUE automatically.

If the *name* parameter contains an empty string, the VI targets the first available function of the same type. If no function is available, *is online* is set to FALSE.

The YModule VI

The `YModule` VI enables you to interface with the "module" section of each Yoctopuce module. It enables you to drive the module led and to know the serial number of the module.

The YModule VI

The *name* input works slightly differently from other VIs. If it is called with a *name* parameter corresponding to a function name, the `YModule` VI finds the *Module* function of the module hosting the function. You can therefore easily find the serial number of the module of any function. This enables you to build the name of other functions which are located on the same module. The following example finds the first available *YHumidity* function and builds the name of the *YTemperature* function located on the same module. The examples provided with the Yoctopuce API make extensive use of this technique.

Using the YModule VI to retrieve functions hosted on the same module

The sensor VIs

All the VIs corresponding to Yoctopuce sensors have exactly the same geometry. Both outputs enable you to retrieve the value measured by the corresponding sensor as well the unit used.

The sensor VIs have all exactly the same geometry

The *update freq* input parameter is a character string enabling you to configure the way in which the output value is updated:

- "auto" : The VI value is updated as soon as the sensor detects a significant modification of the value. It is the default behavior.
- "x/s": The VI value is updated *x* times per second with the current value of the sensor.

- "x/m","x/h": The VI value is updated *x* times per minute (resp. hour) with the average value over the latest period. Note, maximum frequencies are (60/m) and (3600/h), for higher frequencies use the (x/s) syntax.

The update frequency of the VI is a parameter managed by the physical Yoctopuce module. If several VIs try to change the frequency of the same sensor, the valid configuration is that of the latest call. It is however possible to set different update frequencies to different sensors on the same Yoctopuce module.

Changing the update frequency of the same module

The update frequency of the VI is completely independent from the sampling frequency of the sensor, which you usually cannot modify. It is useless and counterproductive to define an update frequency higher than the sensor sampling frequency.

20.5. Functioning and use of VIs

Here is one of the simplest example of VIs using the Yoctopuce API.

Minimal example of use of the LabVIEW Yoctopuce API

This example is based on the `YSensor` VI which is a generic VI enabling you to interface any sensor function of a Yoctopuce module. You can replace this VI by any other from the Yoctopuce API, they all have the same geometry and work in the same way. This example is limited to three actions:

1. It initializes the API in native ("usb") mode with the `YRegisterHub` VI.
2. It displays the value of the first Yoctopuce sensor it finds thanks to the `YSensor` VI.
3. It frees the API thanks to the `YFreeAPI` VI.

This example automatically looks for an available sensor. If there is such a sensor, we can retrieve its name through the *hardware name* output and the *isOnline* output equals TRUE. If there is no available sensor, the VI does not generate an error but emulates a ghost sensor which is "offline". However, if later in the life of the application, a sensor becomes available because it has been connected, *isOnline* switches to TRUE and the *hardware name* contains the name of the sensor. We can therefore easily add a few indicators in the previous example to know how the executions goes.

Use of the hardware name and isOnline outputs

The VIs of the Yoctopuce API are actually an entry door into the library. Internally, this mechanism works independently of the Yoctopuce VIs. Indeed, most communications with electronic modules are managed automatically as background tasks. Therefore, you do not necessarily need to take any specific care to use Yoctopuce VIs, you can for example use them in a non-delayed loop without creating any specific problem for the API.

The Yoctopuce VIs can be used in a non-delayed loop

Note that the `YRegisterHub` VI is not inside the loop. The `YRegisterHub` VI is used to initialize the API. Unless you have several URLs that you need to register, it is better to call the `YRegisterHub` VI only once.

When the *name* parameter is initialized to an empty string, the Yoctopuce VIs automatically look for a function they can work with. This is very handy when you know that there is only one function of the same type available and when you do not want to manage its name. If the *name* parameter contains a hardware name or a logical name, the VI looks for the corresponding function. If it does not find it, it emulates an *offline* function while it waits for the true function to become available.

Using names to identify the functions to be used

Error handling

The LabVIEW Yoctopuce API is coded to handle errors as smoothly as possible: for example, if you use a VI to access a function which does not exist, the *isOnline* output is set to FALSE, the other outputs are set to *NaN*, and thus the inputs do not have any impact. Fatal errors are propagated through the traditional *error in*, *error out* channel.

However, the `YRegisterHub` VI manages connection errors slightly differently. In order to make them easier to manage, connection errors are signaled with *Success* and *error msg* outputs. If there is an issue during a call to the `YRegisterHub` VI, *Success* contains FALSE and *error msg* contains a description of the error.

Error handling

The most common error message is "*Another process is already using yAPI*". It means that another application, LabVIEW or other, already uses the API in native USB mode. For technical reasons, the native USB API can be used by only one application at the same time on the same machine. You can easily work around this limitation by using the network mode.

20.6. Using *Proxy* objects

The Yoctopuce API contains hundreds of methods, functions, and properties. It was not possible, or desirable, to create a VI for each of them. Therefore, there is a VI per class that shows the two properties that Yoctopuce deemed the most useful, but this does not mean that the rest is not available.

Each VI corresponding to a class has two connectors *create ref* and *optional ref* which enable you to obtain a reference on the *Proxy* object of the *.NET Proxy* API on which the LabVIEW library is built.

The connectors to obtain a reference on the Proxy object corresponding to the VI

To obtain this reference, you only need to set *optional ref* to TRUE. Note, it is essential to close all references created in this way, otherwise you risk to quickly saturate the computer memory.

Here is an example which uses this technique to change the luminosity of the leds of a Yoctopuce module.

Regulating the luminosity of the leds of a module

Note that each reference allows you to obtain properties (*property* nodes) as well as methods (*invoke* nodes). By convention, properties are optimized to generate a minimum of communication with the modules. Therefore, we recommend to use them rather than the corresponding *get_xxx* and *set_xxx* methods which might seem equivalent but which are not optimized. Properties also enable you to retrieve the various constants of the API, prefixed with the "_" character. For technical reasons, the *get_xxx* and *set_xxx* methods are not all available as properties.

Property and Invoke nodes: Using properties, methods and constants

You can find a description of all the available properties, functions, and methods in the documentation of the *.NET Proxy* API.

Network mode

On a given machine, there can be only one process accessing local Yoctopuce modules directly by USB (url set to `"usb"`). It is however possible that multiple process connect in parallel to YoctoHubs[7] or tp a machine on which *VirtualHub*[8] is running, including the local machine. Therefore, if you use the local address of your machine (127.0.0.1) and if a VirtualHub runs on it, you can work around the limitation which prevents using the native USB API in parallel.

Network mode

[7] https://www.yoctopuce.com/EN/products/category/extensions-and-networking
[8] www.yoctopuce.com/EN/virtualhub.php

In the same way, there is no limitation on the number of network interfaces to which the API can connect itself in parallel. This means that it is quite possible to make multiple calls to the `YRegisterHub` VI. This is the only case where it is useful to call the `YRegisterHub` VI several times in the life of the application.

You can have multiple network connections

By default, the `YRegisterHub` VI tries to connect itself on the address given as parameter and generates an error (*success*=FALSE) when it cannot do so because nobody answers. But if the *async* parameter is initialized to TRUE, no error is generated when the connection does not succeed. If the connection becomes possible later in the life of the application, the corresponding modules are automatically made available.

Asynchronous connection

20.7. Managing the data logger

Almost all the Yoctopuce sensors have a data logger which enables you to store the measures of the sensors in the non-volatile memory of the module. You can configure the data logger with the VirtualHub, but also with a little bit of LabVIEW code.

Logging

To do so, you must configure the logging frequency by using the "LogFrequency" property which you can reach with a reference on the *Proxy* object of the sensor you are using. Then, you must turn the data logger on thanks to the `YDataLogger` VI. Note that, like with the `YModule` VI, you can obtain the `YDataLogger` VI corresponding to a module with its own name, but also with the name of any of the functions available on the same module.

Activating the data logger

Reading

You can retrieve the data in the data logger with the `YDataLoggerContents` VI.

The YDataLoggerContents VI

Retrieving the data from the logger of a Yoctopuce module is a slow process which can take up to several tens of seconds. Therefore, we designed the VI enabling this operation to work iteratively.

As a first step, you must call the VI with a sensor name, a start date, and an end date (UTC UNIX timestamp). The (0,0) pair enables you to obtain the complete content of the data logger. This first call enables you to obtain a summary of the data logger content and a context.

As a second step, you must call the *YDataLoggerContents* VI in a loop with the context parameter, until the *progress* output reaches the 100 value. At this time, the data output represents the content of the data logger.

Retrieving the content of the data logger

The results and the summary are returned as an array of structures containing the following fields:

- *startTime*: beginning of the measuring period
- *endTime*: end of the measuring period
- *averageValue*: average value for the period
- *minValue*: minimum value over the period
- *maxValue*: maximum value over the period

Note that if the logging frequency is superior to 1Hz, the data logger stores only current values. In this case, *averageValue*, *minValue*, and *maxValue* share the same value.

20.8. Function list

Each VI corresponding to an object of the *Proxy API* enables you to list all the functions of the same class with the *getSimilarfunctions()* method of the corresponding *Proxy* object. Thus, you can easily perform an inventory of all the connected modules, of all the connected sensors, of all the connected relays, and so on.

Retrieving the list of all the modules which are connected

20.9. A word on performances

The LabVIEW Yoctopuce API is optimized so that all the VIs and *.NET Proxy* API object properties generate a minimum of communication with Yoctopuce modules. Thus, you can use them in loops without taking any specific precaution: you *do not have to* slow down the loops with a timer.

These two loops generate little USB communication and do not need to be slowed down

However, almost all the methods of the available Proxy objects initiate a communication with the Yoctopuce modules each time they are called. You should therefore avoid calling them too often without purpose.

This loop, using a method, must be slowed down

20.10. A full example of a LabVIEW program

Here is an example showing how to interface a *ZELIO REG48PUN1RHU* controller with a Yocto-RS485 in LabView. After a call to the *YRegisterHub* VI, the *YSerialPort* VI finds the first serial port available, then gets a reference on the matching *YSerialPortProxy* object. If the serial port is "*online*" then the application reads the *PS* and *SV* registers thanks to the *ModbusWriteRegister* method. Each click on the "+" / "-" button will increment / decrement the *SV* value and update the controller accordingly with a call to the *ModbusWriteRegister* method. When the application is about to close, the *YSerialPortProxy* object reference is closed and the Yoctopuce API is freed with the *YFreeAPI* VI.

Interfacing a ZELIO REG48PUN1RHU controller with a Yocto-RS485 in LabVIEW.

If you read this documentation on screen, you can zoom on the image above. You can also find this example in the LabVIEW Yoctopuce library.

20.11. Differences from other Yoctopuce APIs

Yoctopuce does everything it can to maintain a strong coherence between its different programming libraries. However, LabVIEW being clearly apart as an environment, there are, as a consequence, important differences from the other libraries.

These differences were introduced to make the use of modules as easy as possible and requiring a minimum of LabVIEW code.

YFreeAPI

In the opposite to other languages, you must absolutely free the native API by calling the `YFreeAPI` VI when your code does not need to use the API anymore. If you forget this call, the native API risks to stay locked for the other applications until LabVIEW is completely closed.

Properties

In the opposite to classes of the other APIs, classes available in LabVIEW implement *properties*. By convention, these properties are optimized to generate a minimum of communication with the modules while automatically refreshing. By contrast, methods of type *get_xxx* and *set_xxx* systematically generate communications with the Yoctopuce modules and must be called sparingly.

Callback vs. Properties

There is no callback in the LabVIEW Yoctopuce API, the VIs automatically refresh: they are based on the properties of the *.NET Proxy* API objects.

21. 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.

21.1. 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.

21.2. .NET Assembly

A .NET Assembly enables you to share a set of pre-compiled classes to offer a service, by stating entry points which can be used by third-party applications. In our case, it's the whole Yoctopuce library which is available in the .NET Assembly, so that it can be used in any environment which supports .NET Assembly dynamic loading.

The Yoctopuce library as a .NET Assembly does not contain only the standard C# Yoctopuce library, as this wouldn't have allowed an optimal use in all environments. Indeed, we cannot expect host applications to necessarily offer a thread system or a callback system, although they are very useful to manage plug-and-play events and sensors with a high refresh rate. Likewise, we can't expect from external applications a transparent behavior in cases where a function call in Assembly creates a delay because of network communications.

Therefore, we added to it an additional layer, called *.NET Proxy* library. This additional layer offers an interface very similar to the standard library but somewhat simplified, as it internally manages all the callback mechanisms. Instead, this library offers mirror objects, called *Proxys*, which publish through *Properties* the main attributes of the Yoctopuce functions such as the current measure, configuration parameters, the state, and so on.

.NET Assembly Architecture

The callback mechanism automatically updates the properties of the *Proxys* objects, without the host application needing to care for it. The later can thus, at any time and without any risk of latency, display the value of all properties of Yoctopuce Proxy objects.

Pay attention to the fact that the `yapi.dll` low-level communication library is **not** included in the .NET Assembly. You must therefore keep it together with `DotNetProxyLibrary.dll`. The 32 bit version must be located in the same directory as `DotNetProxyLibrary.dll`, while the 64 bit version must be in a subdirectory `amd64`.

Example of use with MATLAB

Here is how to load our Proxy .NET Assembly in MATLAB and how to read the value of the first sensor connected by USB found on the machine:

```
NET.addAssembly("C:/Yoctopuce/DotNetProxyLibrary.dll");
import YoctoProxyAPI.*

errmsg = YAPIProxy.RegisterHub("usb");
sensor = YSensorProxy.FindSensor("");
measure = sprintf('%.3f %s', sensor.CurrentValue, sensor.Unit);
```

Example of use in PowerShell

PowerShell commands are a little stranger, but we can recognize the same structure:

```
Add-Type -Path "C:/Yoctopuce/DotNetProxyLibrary.dll"

$errmsg = [YoctoProxyAPI.YAPIProxy]::RegisterHub("usb")
$sensor = [YoctoProxyAPI.YSensorProxy]::FindSensor("")
$measure = "{0:n3} {1}" -f $sensor.CurrentValue, $sensor.Unit
```

Specificities of the .NET Proxy library

With regards to classic Yoctopuce libraries, the following differences in particular should be noted:

No FirstModule/nextModule method

To obtain an object referring to the first found module, we call `YModuleProxy.FindModule("")`. If there is no connected module, this method returns an object with its `module.IsOnline` property set to `False`. As soon as a module is connected, the property changes to `True` and the module hardware identifier is updated.

To list modules, you can call the `module.GetSimilarFunctions()` method which returns an array of character strings containing the identifiers of all the modules which were found.

No callback function

Callback functions are implemented internally and they update the object properties. You can therefore simply poll the properties, without significant performance penalties. Be aware that if you

use one of the function that disables callbacks, the automatic refresh of object properties may not work anymore.

A new method `YAPIProxy.GetLog` makes it possible to retrieve low-level debug logs without using callbacks.

Enumerated types
In order to maximize compatibility with host applications, the .NET Proxy library does not use true .NET enumerated types, but simple integers. For each enumerated type, the library includes public constants named according to the possible values. Contrarily to standard Yoctopuce libraries, numeric values always start from `1`, as the value `0` is reserved to return an invalid value, for instance when the device is disconnected.

Invalid numeric results
For all numeric results, rather than using an arbitrary constant, the invalid value returned in case of error is *NaN*. You should therefore use function *isNaN()* to detect this value.

Using .NET Assembly without the Proxy library

If for a reason or another you don't want to use the Proxy library, and if your environment allows it, you can use the standard C# API as it is located in the Assembly, under the `YoctoLib` namespace. Beware however not to mix both types of use: either you go through the Proxy library, or you use he `YoctoLib` version directly, but not both!

CompatibilitÃ©

For the LabVIEW Yoctopuce library to work correctly with your Yoctopuce modules, these modules need to have firmware 37120, or higher.

In order to be compatible with as many versions of Windows as possible, including Windows XP, the *DotNetProxyLibrary.dll* library is compiled in .NET 3.5, which is available by default on all the Windows versions since XP. As of today, we have never met any non-Windows environment able to load a .NET Assembly, so we only ship the low-level communication dll for Windows together with the assembly.

21.3. VirtualHub and HTTP GET

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

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

REST interface

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

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

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

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

Driving a module through the REST interface

Each Yoctopuce module has its own REST interface, available in several variants. Let us imagine a Yocto-RS485 with the *RS485MK1-12345* 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/RS485MK1-12345/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/RS485MK1-12345/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/RS485MK1-12345/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 serialPort function, build the following URL:

```
http://127.0.0.1:4444/bySerial/RS485MK1-12345/api/serialPort.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/RS485MK1-12345/api/serialPort/logicalName
```

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

```
http://127.0.0.1:4444/bySerial/RS485MK1-12345/api/serialPort?logicalName=myFunction
```

You can find the list of available attributes for your Yocto-RS485 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/RS485MK1-12345/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/RS485MK1-12345/dataLogger.json?id=serialPort&utc=1389801080
```

21.4. Using dynamic libraries

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

Filename	Platform
libyapi.dylib	Max OS X
libyapi-amd64.so	Linux Intel (64 bits)
libyapi-armel.so	Linux ARM EL (32 bits)
libyapi-armhf.so	Linux ARM HL (32 bits)
libyapi-aarch64.so	Linux ARM (64 bits)
libyapi-i386.so	Linux Intel (32 bits)
yapi64.dll	Windows (64 bits)
yapi.dll	Windows (32 bits)

These dynamic libraries contain all the functions necessary to completely rebuild the whole high level API in any language able to integrate these libraries. This chapter nevertheless restrains itself to describing basic use of the modules.

Driving a module

The three essential functions of the low level API are the following:

```
int yapiInitAPI(int connection_type, char *errmsg);
int yapiUpdateDeviceList(int forceupdate, char *errmsg);
int yapiHTTPRequest(char *device, char *request, char* buffer,int buffsize,int *fullsize,
char *errmsg);
```

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

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

Finally, the *yapiHTTPRequest* function sends HTTP requests to the module REST API. The *device* parameter contains the serial number or the logical name of the module which you want to reach. The *request* parameter contains the full HTTP request (including terminal line breaks). *buffer* points to a character buffer long enough to contain the answer. *buffsize* is the size of the buffer. *fullsize* is a pointer to an integer to which will be assigned the actual size of the answer. The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message. This pointer can also point to *null*. The function returns a negative integer in case of error, zero otherwise.

The format of the requests is the same as the one described in the *VirtualHub et HTTP GET* section. All the character strings used by the API are strings made of 8-bit characters: Unicode and UTF8 are not supported.

The resutlt returned in the buffer variable respects the HTTP protocol. It therefore includes an HTTP header. This header ends with two empty lines, that is a sequence of four ASCII characters 13, 10, 13, 10.

Here is a sample program written in pascal using the *yapi.dll* DLL to read and then update the luminosity of a module.

```
// Dll functions import
function  yapiInitAPI(mode:integer;
                      errmsg : pansichar):integer;cdecl;
```

```
                      external 'yapi.dll' name 'yapiInitAPI';
function  yapiUpdateDeviceList(force:integer;errmsg : pansichar):integer;cdecl;
                      external 'yapi.dll' name 'yapiUpdateDeviceList';
function  yapiHTTPRequest(device:pansichar;url:pansichar; buffer:pansichar;
                      buffsize:integer;var fullsize:integer;
                      errmsg : pansichar):integer;cdecl;
                      external 'yapi.dll' name 'yapiHTTPRequest';

var
 errmsgBuffer  : array [0..256] of ansichar;
 dataBuffer    : array [0..1024] of ansichar;
 errmsg,data   : pansichar;
 fullsize,p    : integer;

const
  serial      = 'RS485MK1-12345';
  getValue = 'GET /api/module/luminosity HTTP/1.1'#13#10#13#10;
  setValue = 'GET /api/module?luminosity=100 HTTP/1.1'#13#10#13#10;

begin
  errmsg  :=  @errmsgBuffer;
  data    :=  @dataBuffer;
  // API  initialization
  if(yapiInitAPI(1,errmsg)<0) then
   begin
    writeln(errmsg);
    halt;
  end;

  // forces a device inventory
  if( yapiUpdateDeviceList(1,errmsg)<0) then
    begin
     writeln(errmsg);
     halt;
   end;

  // requests the  module luminosity
  if (yapiHTTPRequest(serial,getValue,data,sizeof(dataBuffer),fullsize,errmsg)<0) then
   begin
     writeln(errmsg);
     halt;
   end;

  // searches for the HTTP header end
  p := pos(#13#10#13#10,data);

  // displays the response minus the HTTP header
  writeln(copy(data,p+4,length(data)-p-3));

  // changes the luminosity
  if (yapiHTTPRequest(serial,setValue,data,sizeof(dataBuffer),fullsize,errmsg)<0) then
   begin
     writeln(errmsg);
     halt;
   end;

end.
```

Module inventory

To perform an inventory of Yoctopuce modules, you need two functions from the dynamic library:

```
int yapiGetAllDevices(int *buffer,int maxsize,int *neededsize,char *errmsg);
int yapiGetDeviceInfo(int devdesc,yDeviceSt *infos, char *errmsg);
```

The *yapiGetAllDevices* function retrieves the list of all connected modules as a list of handles. *buffer* points to a 32-bit integer array which contains the returned handles. *maxsize* is the size in bytes of the buffer. To *neededsize* is assigned the necessary size to store all the handles. From this, you can deduce either the number of connected modules or that the input buffer is too small. The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message. This pointer can also point to *null*. The function returns a negative integer in case of error, zero otherwise.

The *yapiGetDeviceInfo* function retrieves the information related to a module from its handle. *devdesc* is a 32-bit integer representing the module and which was obtained through *yapiGetAllDevices*. *infos* points to a data structure in which the result is stored. This data structure has the following format:

Name	Type	Size (bytes)	Description
vendorid	int	4	Yoctopuce USB ID
deviceid	int	4	Module USB ID
devrelease	int	4	Module version
nbinbterfaces	int	4	Number of USB interfaces used by the module
manufacturer	char[]	20	Yoctopuce (null terminated)
productname	char[]	28	Model (null terminated)
serial	char[]	20	Serial number (null terminated)
logicalname	char[]	20	Logical name (null terminated)
firmware	char[]	22	Firmware version (null terminated)
beacon	byte	1	Beacon state (0/1)

The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message.

Here is a sample program written in pascal using the *yapi.dll* DLL to list the connected modules.

```
// device description structure
type yDeviceSt = packed record
   vendorid        : word;
   deviceid        : word;
   devrelease      : word;
   nbinbterfaces   : word;
   manufacturer    : array [0..19] of ansichar;
   productname     : array [0..27] of ansichar;
   serial          : array [0..19] of ansichar;
   logicalname     : array [0..19] of ansichar;
   firmware        : array [0..21] of ansichar;
   beacon          : byte;
 end;

// Dll function import
function  yapiInitAPI(mode:integer;
                      errmsg : pansichar):integer;cdecl;
                      external 'yapi.dll' name 'yapiInitAPI';

function  yapiUpdateDeviceList(force:integer;errmsg : pansichar):integer;cdecl;
                      external 'yapi.dll' name 'yapiUpdateDeviceList';

function  yapiGetAllDevices( buffer:pointer;
                             maxsize:integer;
                             var neededsize:integer;
                             errmsg : pansichar):integer; cdecl;
                             external 'yapi.dll' name 'yapiGetAllDevices';

function  apiGetDeviceInfo(d:integer; var infos:yDeviceSt;
                             errmsg : pansichar):integer;  cdecl;
                             external 'yapi.dll' name 'yapiGetDeviceInfo';


var
 errmsgBuffer  : array [0..256] of ansichar;
 dataBuffer    : array [0..127] of integer;   // max of 128 USB devices
 errmsg,data   : pansichar;
 neededsize,i  : integer;
 devinfos      : yDeviceSt;

begin
  errmsg  :=  @errmsgBuffer;

  // API  initialization
  if(yapiInitAPI(1,errmsg)<0) then
   begin
    writeln(errmsg);
    halt;
  end;
```

```
   // forces a device inventory
  if( yapiUpdateDeviceList(1,errmsg)<0) then
    begin
     writeln(errmsg);
     halt;
   end;

  // loads all device handles into dataBuffer
  if yapiGetAllDevices(@dataBuffer,sizeof(dataBuffer),neededsize,errmsg)<0 then
    begin
     writeln(errmsg);
     halt;
    end;

  // gets device info from each handle
  for i:=0 to  neededsize div sizeof(integer)-1 do
   begin
     if (apiGetDeviceInfo(dataBuffer[i], devinfos, errmsg)<0) then
       begin
         writeln(errmsg);
         halt;
       end;
     writeln(pansichar(@devinfos.serial)+' ('+pansichar(@devinfos.productname)+')');
   end;

end.
```

VB6 and yapi.dll

Each entry point from the yapi.dll is duplicated. You will find one regular C-decl version and one Visual Basic 6 compatible version, prefixed with *vb6_*.

21.5. Porting the high level library

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

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

22. Advanced programming

The preceding chapters have introduced, in each available language, the basic programming functions which can be used with your Yocto-RS485 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.

22.1. 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 t he 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_unt()` to get the physical unit of the sensor, if you need to display it.

A complete example
You can find a complete example implemented in your favorite programming language in the *Examples/Prog-EventBased* directory of the corresponding library.

22.2. The data logger

Your Yocto-RS485 is equipped with a data logger able to store non-stop the measures performed by the module. The maximal frequency is 100 times per second (i.e. "100/s"), and the minimal frequency is 1 time per hour (i.e. "1/h"). When the frequency is higher than or equal to 1/s, the measure represents an instant value. When the frequency is below, the measure will include distinct minimal, maximal and average values based on a sampling performed automatically by the device.

Note that is useless and counter-productive to set a recording frequency higher than the native sampling frequency of the recorded sensor.

The data logger flash memory can store about 500'000 instant measures, or 125'000 averaged measures. When the memory is about to be saturated, the oldest measures are automatically erased.

Make sure not to leave the data logger running at high speed unless really needed: the flash memory can only stand a limited number of erase cycles (typically 100'000 cycles). When running at full speed, the datalogger can burn more than 100 cycles per day ! Also be aware that it is useless to record measures at a frequency higher than the refresh frequency of the physical sensor itself.

Starting/stopping the datalogger

The data logger can be started with the `set_recording()` method.

```
YDataLogger l = YDataLogger.FirstDataLogger();
l.set_recording(YDataLogger.RECORDING_ON);
```

It is possible to make the data recording start automatically as soon as the module is powered on.

```
YDataLogger l = YDataLogger.FirstDataLogger();
l.set_autoStart(YDataLogger.AUTOSTART_ON);
l.get_module().saveToFlash();  // do not forget to save the setting
```

Note: Yoctopuce modules do not need an active USB connection to work: they start working as soon as they are powered on. The Yocto-RS485 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-RS485 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-RS485 flash memory, you must call the `get_recordedData()` method of the desired sensor, and specify the time interval for which you want to retrieve measures. The time interval is given by the start and stop UNIX timestamp. You can also specify 0 if you don't want any start or stop limit.

The `get_recordedData()` method does not return directly am array of measured values, since in some cases it would cause a huge load that could affect the responsiveness of the application. Instead, this function will return an `YDataSet` object that can be used to retrieve immediately an overview of the measured data (summary), and then to load progressively the details when desired.

Here are the main methods used to retrieve recorded measures:

1. **dataset = sensor.get_recordedData(0,0)**: select the desired time interval
2. **dataset.loadMore()**: load data from the device, progressively
3. **dataset.get_summary()**: get a single measure summarizing the full time interval
4. **dataset.get_preview()**: get an array of measures representing a condensed version of the whole set of measures on the selected time interval (reduced by a factor of approx. 200)
5. **dataset.get_measures()**: get an array with all detailled measures (that grows while `loadMore` is being called repeteadly)

Measures are instances of `YMeasure` [2]. They store simultaneously the minimal, average and maximal value at a given time, that you can retrieve using methods **get_minValue()**, **get_averageValue()** and **get_maxValue()** respectively. Here is a small example that uses the functions above:

```
// We will retrieve all measures, without time limit
YDataSet dataset = sensor.get_recordedData(0, 0);

// First call to loadMore() loads the summary/preview
dataset.loadMore();
YMeasure summary = dataset.get_summary();
```

[2] The `YMeasure` objects used by the data logger are exactly the same kind as those passed as argument to the timed report callbacks.

```
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()));

// 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-RS485 does not have a battery, it cannot guess alone the current time when powered on. Nevertheless, the Yocto-RS485 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-RS485 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-RS485 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-RS485 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-RS485 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-RS485 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-RS485 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-RS485 might be be subject to a clock skew (theoretically up to 2%).

22.3. Sensor calibration

Your Yocto-RS485 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-RS485. 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-RS485 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-RS485. 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-RS485 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*.

Measure correction with 1 calibration point, here (5,10)

The application is very simple: you only need to call the *calibrateFromPoints()* method of the function you wish to correct. The following code applies the correction illustrated on the graph above to the first 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.

Measure correction with the two calibration points (10,5) and (25,10).

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

```
Double[] ValuesBefore = {10,25};
Double[] ValuesAfter  = {5,10};
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.

Correction example with 3 calibration points

Back to normal
To cancel the effect of a calibration on a function, call the *calibrateFromPoints()* method with two empty arrays.

```
Double[] ValuesBefore = {};
Double[] ValuesAfter  = {};
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.

Limitations

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

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

Arbitrary interpolation

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

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

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

23. Firmware Update

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

23.1. The VirtualHub or the YoctoHub

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

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

23.2. The command line library

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

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

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

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

23.3. The Android application Yocto-Firmware

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

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

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

23.4. Updating the firmware with the programming library

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

Saving and restoring parameters

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

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

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

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

Finding the correct firmware

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

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

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

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

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

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

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

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

Updating the firmware

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

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

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

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

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

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

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

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

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

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

An Android characteristic

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

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

23.5. The "update" mode

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

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

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

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

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

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

24. High-level API Reference

This chapter summarizes the high-level API functions to drive your Yocto-RS485. 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.

24.1. Class YAPI

General functions

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

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

java	import com.yoctopuce.YoctoAPI.YAPI;
dnp	import YoctoProxyAPI.YAPIProxy
cp	#include "yocto_api_proxy.h"
ml	import YoctoProxyAPI.YAPIProxy"
js	<script type='text/javascript' src='yocto_api.js'></script>
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
uwp	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *
php	require_once('yocto_api.php');
es	in HTML: <script src="../../lib/yocto_api.js"></script> in node.js: require('yoctolib-es2017/yocto_api.js');
vi	YModule.vi

Global functions
YAPI.CheckLogicalName(**name**) Checks if a given string is valid as logical name for a module or a function.
YAPI.ClearHTTPCallbackCacheDir(**bool_removeFiles**) Disables the HTTP callback cache.
YAPI.DisableExceptions() Disables the use of exceptions to report runtime errors.
YAPI.EnableExceptions() Re-enables the use of exceptions for runtime error handling.
YAPI.EnableUSBHost(**osContext**) This function is used only on Android.
YAPI.FreeAPI() Waits for all pending communications with Yoctopuce devices to be completed then frees dynamically allocated resources used by the Yoctopuce library.
YAPI.GetAPIVersion() Returns the version identifier for the Yoctopuce library in use.
YAPI.GetCacheValidity() Returns the validity period of the data loaded by the library.
YAPI.GetDeviceListValidity() Returns the delay between each forced enumeration of the used YoctoHubs.
YAPI.GetDllArchitecture()

Returns the system architecture for the Yoctopuce communication library in use.

YAPI.GetDllPath()

Returns the paths of the DLLs for the Yoctopuce library in use.

YAPI.GetLog(lastLogLine)

Retrieves Yoctopuce low-level library diagnostic logs.

YAPI.GetNetworkTimeout()

Returns the network connection delay for `yRegisterHub()` and `yUpdateDeviceList()`.

YAPI.GetTickCount()

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

YAPI.HandleEvents(errmsg)

Maintains the device-to-library communication channel.

YAPI.InitAPI(mode, errmsg)

Initializes the Yoctopuce programming library explicitly.

YAPI.PreregisterHub(url, errmsg)

Fault-tolerant alternative to `yRegisterHub()`.

YAPI.RegisterDeviceArrivalCallback(arrivalCallback)

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

YAPI.RegisterDeviceRemovalCallback(removalCallback)

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

YAPI.RegisterHub(url, errmsg)

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

YAPI.RegisterHubDiscoveryCallback(hubDiscoveryCallback)

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

YAPI.RegisterHubWebsocketCallback(ws, errmsg, authpwd)

Variant to `yRegisterHub()` used to initialize Yoctopuce API on an existing Websocket session, as happens for incoming WebSocket callbacks.

YAPI.RegisterLogFunction(logfun)

Registers a log callback function.

YAPI.SelectArchitecture(arch)

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

YAPI.SetCacheValidity(cacheValidityMs)

Change the validity period of the data loaded by the library.

YAPI.SetDelegate(object)

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

YAPI.SetDeviceListValidity(deviceListValidity)

Modifies the delay between each forced enumeration of the used YoctoHubs.

YAPI.SetHTTPCallbackCacheDir(str_directory)

Enables the HTTP callback cache.

YAPI.SetNetworkTimeout(networkMsTimeout)

Modifies the network connection delay for `yRegisterHub()` and `yUpdateDeviceList()`.

YAPI.SetTimeout(callback, ms_timeout, args)

Invoke the specified callback function after a given timeout.

YAPI.SetUSBPacketAckMs(pktAckDelay)

Enables the acknowledge of every USB packet received by the Yoctopuce library.

YAPI.Sleep(ms_duration, errmsg)

Pauses the execution flow for a specified duration.

YAPI.TestHub(**url**, **mstimeout**, **errmsg**)

Test if the hub is reachable.

YAPI.TriggerHubDiscovery(**errmsg**)

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

YAPI.UnregisterHub(**url**)

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

YAPI.UpdateDeviceList(**errmsg**)

Triggers a (re)detection of connected Yoctopuce modules.

YAPI.UpdateDeviceList_async(**callback**, **context**)

Triggers a (re)detection of connected Yoctopuce modules.

YAPI.CheckLogicalName() YAPI.CheckLogicalName() YAPI

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

```
js    function yCheckLogicalName( name)
cpp   bool yCheckLogicalName( string name)
m     +(BOOL) CheckLogicalName :(NSString *) name
pas   boolean yCheckLogicalName( name: string): boolean
vb    function yCheckLogicalName( ByVal name As String) As Boolean
cs    static bool CheckLogicalName( string name)
java  boolean CheckLogicalName( String name)
uwp   bool CheckLogicalName( string name)
py    CheckLogicalName( name)
php   function yCheckLogicalName( $name)
es    async CheckLogicalName( name)
```

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

Parameters :

name a string containing the name to check.

Returns :

`true` if the name is valid, `false` otherwise.

YAPI.ClearHTTPCallbackCacheDir() YAPI

YAPI.ClearHTTPCallbackCacheDir()

Disables the HTTP callback cache.

```
php function yClearHTTPCallbackCacheDir( $bool_removeFiles)
```

This method disables the HTTP callback cache, and can additionally cleanup the cache directory.

Parameters :

bool_removeFiles True to clear the content of the cache.

Returns :

nothing.

YAPI.DisableExceptions() YAPI

YAPI.DisableExceptions()

Disables the use of exceptions to report runtime errors.

js	function **yDisableExceptions**( )
cpp	void **yDisableExceptions**( )
m	+(void) **DisableExceptions**
pas	**yDisableExceptions**( )
vb	procedure **yDisableExceptions**( )
cs	static void **DisableExceptions**( )
uwp	void **DisableExceptions**( )
py	**DisableExceptions**( )
php	function **yDisableExceptions**( )
es	async **DisableExceptions**( )

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

YAPI.EnableExceptions() YAPI

YAPI.EnableExceptions()

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

js	function **yEnableExceptions**( )
cpp	void **yEnableExceptions**( )
m	+(void) **EnableExceptions**
pas	**yEnableExceptions**( )
vb	procedure **yEnableExceptions**( )
cs	static void **EnableExceptions**( )
uwp	void **EnableExceptions**( )
py	**EnableExceptions**( )
php	function **yEnableExceptions**( )
es	async **EnableExceptions**( )

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

YAPI.EnableUSBHost() YAPI

YAPI.EnableUSBHost()

This function is used only on Android.

```
java void EnableUSBHost( Object osContext)
```

Before calling `yRegisterHub("usb")` you need to activate the USB host port of the system. This function takes as argument, an object of class android.content.Context (or any subclass). It is not necessary to call this function to reach modules through the network.

Parameters :

osContext an object of class android.content.Context (or any subclass).

YAPI.FreeAPI() YAPI

YAPI.FreeAPI()

Waits for all pending communications with Yoctopuce devices to be completed then frees dynamically allocated resources used by the Yoctopuce library.

js	function **yFreeAPI**( )
cpp	void **yFreeAPI**( )
m	+(void) **FreeAPI**
pas	**yFreeAPI**( )
vb	procedure **yFreeAPI**( )
cs	static void **FreeAPI**( )
java	void **FreeAPI**( )
uwp	void **FreeAPI**( )
py	**FreeAPI**( )
php	function **yFreeAPI**( )
es	async **FreeAPI**( )
dnp	static void **FreeAPI**( )
cp	static void **FreeAPI**( )

From an operating system standpoint, it is generally not required to call this function since the OS will automatically free allocated resources once your program is completed. However there are two situations when you may really want to use that function: - Free all dynamically allocated memory blocks in order to track a memory leak. - Send commands to devices right before the end of the program. Since commands are sent in an asynchronous way the program could exit before all commands are effectively sent. You should not call any other library function after calling `yFreeAPI()`, or your program will crash.

YAPI.GetAPIVersion() YAPI.GetAPIVersion() — YAPI

Returns the version identifier for the Yoctopuce library in use.

js	function **yGetAPIVersion**( )
cpp	string **yGetAPIVersion**( )
m	+(NSString*) **GetAPIVersion**
pas	string **yGetAPIVersion**( ): string
vb	function **yGetAPIVersion**( ) As String
cs	static String **GetAPIVersion**( )
java	static String **GetAPIVersion**( )
uwp	static string **GetAPIVersion**( )
py	**GetAPIVersion**( )
php	function **yGetAPIVersion**( )
es	async **GetAPIVersion**( )
dnp	static string **GetAPIVersion**( )
cp	static string **GetAPIVersion**( )

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

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

Returns :
a character string describing the library version.

YAPI.GetCacheValidity() YAPI.GetCacheValidity() YAPI

Returns the validity period of the data loaded by the library.

cpp	static u64 **yGetCacheValidity**( )
m	+(u64) **GetCacheValidity**
pas	u64 **yGetCacheValidity**( ): u64
vb	function **yGetCacheValidity**( ) As Long
cs	ulong **GetCacheValidity**( )
java	long **GetCacheValidity**( )
uwp	async Task<ulong> **GetCacheValidity**( )
py	**GetCacheValidity**( )
php	function **yGetCacheValidity**( )
es	async **GetCacheValidity**( )

This method returns the cache validity of all attributes module functions. Note: This function must be called after `yInitAPI` .

Returns :
an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

YAPI.GetDeviceListValidity() YAPI

YAPI.GetDeviceListValidity()

Returns the delay between each forced enumeration of the used YoctoHubs.

cpp	static int **yGetDeviceListValidity**( )
m	+(int) **GetDeviceListValidity**
pas	LongInt **yGetDeviceListValidity**( ): LongInt
vb	function **yGetDeviceListValidity**( ) As Integer
cs	int **GetDeviceListValidity**( )
java	int **GetDeviceListValidity**( )
uwp	async Task<int> **GetDeviceListValidity**( )
py	**GetDeviceListValidity**( )
php	function **yGetDeviceListValidity**( )
es	async **GetDeviceListValidity**( )

Note: you must call this function after `yInitAPI`.

Returns :
the number of seconds between each enumeration.

YAPI.GetDllArchitecture() YAPI

YAPI.GetDllArchitecture()

Returns the system architecture for the Yoctopuce communication library in use.

dnp static string **GetDllArchitecture**( )

On Windows, the architecture can be `"Win32"` or `"Win64"`. On ARM machines, the architecture is `"Armhf32"` or `"Aarch64"`. On other Linux machines, the architecture is `"Linux32"` or `"Linux64"`. On MacOS, the architecture is `"MacOs32"` or `"MacOs64"`.

Returns :
a character string describing the system architecture of the low-level communication library.

YAPI.GetDllPath() YAPI

YAPI.GetDllPath()

Returns the paths of the DLLs for the Yoctopuce library in use.

dnp static string **GetDllPath**( )

For architectures that require multiple DLLs, in particular when using a .NET assembly DLL, the returned string takes the form `"DotNetProxy=/...; yapi=/...;"`, where the first path corresponds to the .NET assembly DLL and the second path corresponds to the low-level communication library.

Returns :
a character string describing the DLL path.

YAPI.GetLog() YAPI.GetLog() — YAPI

Retrieves Yoctopuce low-level library diagnostic logs.

dnp static string **GetLog**(string **lastLogLine**)

cp static string **GetLog**(**string**)

This method allows to progessively retrieve API logs. The interface is line-based: it must called it within a loop until the returned value is an empty string. Make sure to exit the loop when an empty string is returned, as feeding an empty string into the `lastLogLine` parameter for the next call would restart enumerating logs from the oldest message available.

Parameters :

lastLogLine On first call, provide an empty string. On subsequent calls, provide the last log line returned by `GetLog()`.

Returns :

a string with the log line immediately following the one given in argument, if such line exist. Returns an empty string otherwise, when completed.

YAPI.GetNetworkTimeout() YAPI

YAPI.GetNetworkTimeout()

Returns the network connection delay for `yRegisterHub()` and `yUpdateDeviceList()`.

cpp	static int **yGetNetworkTimeout**( )
m	+(int) **GetNetworkTimeout**
pas	LongInt **yGetNetworkTimeout**( ): LongInt
vb	function **yGetNetworkTimeout**( ) As Integer
cs	int **GetNetworkTimeout**( )
java	int **GetNetworkTimeout**( )
uwp	async Task<int> **GetNetworkTimeout**( )
py	**GetNetworkTimeout**( )
php	function **yGetNetworkTimeout**( )
es	async **GetNetworkTimeout**( )
dnp	static int **GetNetworkTimeout**( )
cp	static int **GetNetworkTimeout**( )

This delay impacts only the YoctoHubs and VirtualHub which are accessible through the network. By default, this delay is of 20000 milliseconds, but depending or you network you may want to change this delay, for example if your network infrastructure is based on a GSM connection.

Returns :
the network connection delay in milliseconds.

YAPI.GetTickCount() YAPI

YAPI.GetTickCount()

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

js	function **yGetTickCount**( )
cpp	u64 **yGetTickCount**( )
m	+(u64) **GetTickCount**
pas	u64 **yGetTickCount**( ): u64
vb	function **yGetTickCount**( ) As Long
cs	static ulong **GetTickCount**( )
java	static long **GetTickCount**( )
uwp	static ulong **GetTickCount**( )
py	**GetTickCount**( )
php	function **yGetTickCount**( )
es	**GetTickCount**( )

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

Returns :
a long integer corresponding to the millisecond counter.

YAPI.HandleEvents() YAPI.HandleEvents() YAPI

Maintains the device-to-library communication channel.

```
js    function yHandleEvents( errmsg)
cpp   YRETCODE yHandleEvents( string errmsg)
m     +(YRETCODE) HandleEvents :(NSError**) errmsg
pas   integer yHandleEvents( var errmsg: string): integer
vb    function yHandleEvents( ByRef errmsg As String) As YRETCODE
cs    static YRETCODE HandleEvents( ref string errmsg)
java  int HandleEvents( )
uwp   async Task<int> HandleEvents( )
py    HandleEvents( errmsg=None)
php   function yHandleEvents( &$errmsg)
es    async HandleEvents( errmsg)
```

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

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

Parameters :

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

YAPI.InitAPI() YAPI

YAPI.InitAPI()

Initializes the Yoctopuce programming library explicitly.

js	function **yInitAPI**(**mode**, **errmsg**)
cpp	YRETCODE **yInitAPI**(int **mode**, string **errmsg**)
m	+(YRETCODE) **InitAPI** :(int) **mode** :(NSError**) **errmsg**
pas	integer **yInitAPI**(**mode**: integer, var **errmsg**: string): integer
vb	function **yInitAPI**(ByVal **mode** As Integer, ByRef **errmsg** As String) As Integer
cs	static int **InitAPI**(int **mode**, ref string **errmsg**)
java	int **InitAPI**(int **mode**)
uwp	async Task<int> **InitAPI**(int **mode**)
py	**InitAPI**(**mode**, **errmsg**=None)
php	function **yInitAPI**(**$mode**, **&$errmsg**)
es	async **InitAPI**(**mode**, **errmsg**)

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

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

Parameters :

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

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

YAPI.PreregisterHub() YAPI.PreregisterHub() YAPI

Fault-tolerant alternative to `yRegisterHub()`.

```
js    function yPreregisterHub( url, errmsg)
cpp   YRETCODE yPreregisterHub( string url, string errmsg)
m     +(YRETCODE) PreregisterHub :(NSString *) url :(NSError**) errmsg
pas   integer yPreregisterHub( url: string, var errmsg: string): integer
vb    function yPreregisterHub( ByVal url As String,
                                ByRef errmsg As String) As Integer
cs    static int PreregisterHub( string url, ref string errmsg)
java  int PreregisterHub( String url)
uwp   async Task<int> PreregisterHub( string url)
py    PreregisterHub( url, errmsg=None)
php   function yPreregisterHub( $url, &$errmsg)
es    async PreregisterHub( url, errmsg)
dnp   static string PreregisterHub( string url)
cp    static string PreregisterHub( string)
```

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

Parameters :

url a string containing either **"usb"**,**"callback"** or the root URL of the hub to monitor

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

YAPI.RegisterDeviceArrivalCallback() YAPI

YAPI.RegisterDeviceArrivalCallback()

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

```
js    function yRegisterDeviceArrivalCallback( arrivalCallback)
cpp   void yRegisterDeviceArrivalCallback( yDeviceUpdateCallback arrivalCallback)
m     +(void) RegisterDeviceArrivalCallback :(yDeviceUpdateCallback) arrivalCallback
pas   yRegisterDeviceArrivalCallback( arrivalCallback: yDeviceUpdateFunc)
vb    procedure yRegisterDeviceArrivalCallback( ByVal arrivalCallback As yDeviceUpdateFunc)
cs    static void RegisterDeviceArrivalCallback( yDeviceUpdateFunc arrivalCallback)
java  void RegisterDeviceArrivalCallback( DeviceArrivalCallback arrivalCallback)
uwp   void RegisterDeviceArrivalCallback( DeviceUpdateHandler arrivalCallback)
py    RegisterDeviceArrivalCallback( arrivalCallback)
php   function yRegisterDeviceArrivalCallback( $arrivalCallback)
es    async RegisterDeviceArrivalCallback( arrivalCallback)
```

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

Parameters :

arrivalCallback a procedure taking a `YModule` parameter, or `null`

YAPI.RegisterDeviceRemovalCallback() YAPI

YAPI.RegisterDeviceRemovalCallback()

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

js	function **yRegisterDeviceRemovalCallback**(**removalCallback**)
cpp	void **yRegisterDeviceRemovalCallback**(yDeviceUpdateCallback **removalCallback**)
m	+(void) **RegisterDeviceRemovalCallback** :(yDeviceUpdateCallback) **removalCallback**
pas	**yRegisterDeviceRemovalCallback**(**removalCallback**: yDeviceUpdateFunc)
vb	procedure **yRegisterDeviceRemovalCallback**(ByVal **removalCallback** As yDeviceUpdateFunc)
cs	static void **RegisterDeviceRemovalCallback**(yDeviceUpdateFunc **removalCallback**)
java	void **RegisterDeviceRemovalCallback**(DeviceRemovalCallback **removalCallback**)
uwp	void **RegisterDeviceRemovalCallback**(DeviceUpdateHandler **removalCallback**)
py	**RegisterDeviceRemovalCallback**(**removalCallback**)
php	function **yRegisterDeviceRemovalCallback**(**$removalCallback**)
es	async **RegisterDeviceRemovalCallback**(**removalCallback**)

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

Parameters :

removalCallback a procedure taking a `YModule` parameter, or `null`

YAPI.RegisterHub() YAPI

YAPI.RegisterHub()

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

```
js    function yRegisterHub( url, errmsg)
cpp   YRETCODE yRegisterHub( string url, string errmsg)
m     +(YRETCODE) RegisterHub :(NSString *) url :(NSError**) errmsg
pas   integer yRegisterHub( url: string, var errmsg: string): integer
vb    function yRegisterHub( ByVal url As String,
                             ByRef errmsg As String) As Integer
cs    static int RegisterHub( string url, ref string errmsg)
java  int RegisterHub( String url)
uwp   async Task<int> RegisterHub( string url)
py    RegisterHub( url, errmsg=None)
php   function yRegisterHub( $url, &$errmsg)
es    async RegisterHub( url, errmsg)
dnp   static string RegisterHub( string url)
cp    static string RegisterHub( string)
```

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

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

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

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

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

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

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

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

Parameters :

url a string containing either **"usb"**,**"callback"** or the root URL of the hub to monitor

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

Returns :
`YAPI_SUCCESS` when the call succeeds.

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

YAPI.RegisterHubDiscoveryCallback() YAPI.RegisterHubDiscoveryCallback() YAPI

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

cpp	void **yRegisterHubDiscoveryCallback**(YHubDiscoveryCallback **hubDiscoveryCallback**)
m	+(void) **RegisterHubDiscoveryCallback** : (YHubDiscoveryCallback) **hubDiscoveryCallback**
pas	**yRegisterHubDiscoveryCallback**(**hubDiscoveryCallback**: YHubDiscoveryCallback)
vb	procedure **yRegisterHubDiscoveryCallback**(ByVal **hubDiscoveryCallback** As YHubDiscoveryCallback)
cs	static void **RegisterHubDiscoveryCallback**(YHubDiscoveryCallback **hubDiscoveryCallback**)
java	void **RegisterHubDiscoveryCallback**(HubDiscoveryCallback **hubDiscoveryCallback**)
uwp	async Task **RegisterHubDiscoveryCallback**(HubDiscoveryHandler **hubDiscoveryCallback**)
py	**RegisterHubDiscoveryCallback**(**hubDiscoveryCallback**)
es	async **RegisterHubDiscoveryCallback**(**hubDiscoveryCallback**)

The callback has two string parameter, the first one contain the serial number of the hub and the second contain the URL of the network hub (this URL can be passed to RegisterHub). This callback will be invoked while yUpdateDeviceList is running. You will have to call this function on a regular basis.

Parameters :

hubDiscoveryCallback a procedure taking two string parameter, the serial

YAPI.RegisterHubWebsocketCallback() YAPI

YAPI.RegisterHubWebsocketCallback()

Variant to `yRegisterHub()` used to initialize Yoctopuce API on an existing Websocket session, as happens for incoming WebSocket callbacks.

Parameters :

ws	node WebSocket object for the incoming WebSocket callback connection
errmsg	a string passed by reference to receive any error message.
authpwd	the optional authentication password, required only authentication is configured on the calling hub.

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

YAPI.RegisterLogFunction() YAPI.RegisterLogFunction() YAPI

Registers a log callback function.

cpp	void **yRegisterLogFunction**(yLogFunction **logfun**)
m	+(void) **RegisterLogFunction** :(yLogCallback) **logfun**
pas	**yRegisterLogFunction**(**logfun**: yLogFunc)
vb	procedure **yRegisterLogFunction**(ByVal **logfun** As yLogFunc)
cs	static void **RegisterLogFunction**(yLogFunc **logfun**)
java	void **RegisterLogFunction**(LogCallback **logfun**)
uwp	void **RegisterLogFunction**(LogHandler **logfun**)
py	**RegisterLogFunction**(**logfun**)
es	async **RegisterLogFunction**(**logfun**)

This callback will be called each time the API have something to say. Quite useful to debug the API.

Parameters :

logfun a procedure taking a string parameter, or `null`

YAPI.SelectArchitecture() YAPI.SelectArchitecture() YAPI

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

py **SelectArchitecture(arch)**

By default, the Python library automatically detects the appropriate library to use. However, for Linux ARM, it not possible to reliably distinguish between a Hard Float (armhf) and a Soft Float (armel) install. For in this case, it is therefore recommended to manually select the proper architecture by calling `SelectArchitecture()` before any other call to the library.

Parameters :

arch A string containing the architecture to use. Possibles value are: `"armhf"`,`"armel"`, `"aarch64"`,`"i386"`,`"x86_64"`, `"32bit"`, `"64bit"`

Returns :

nothing.

On failure, throws an exception.

YAPI.SetCacheValidity() YAPI.SetCacheValidity() — YAPI

Change the validity period of the data loaded by the library.

cpp	static void **ySetCacheValidity**(u64 **cacheValidityMs**)
m	+(void) **SetCacheValidity** : (u64) **cacheValidityMs**
pas	**ySetCacheValidity**(**cacheValidityMs**: u64)
vb	procedure **ySetCacheValidity**(ByVal **cacheValidityMs** As Long)
cs	void **SetCacheValidity**(ulong **cacheValidityMs**)
java	void **SetCacheValidity**(long **cacheValidityMs**)
uwp	async Task **SetCacheValidity**(ulong **cacheValidityMs**)
py	**SetCacheValidity**(**cacheValidityMs**)
php	function **ySetCacheValidity**(**$cacheValidityMs**)
es	async **SetCacheValidity**(**cacheValidityMs**)

By default, when accessing a module, all the attributes of the module functions are automatically kept in cache for the standard duration (5 ms). This method can be used to change this standard duration, for example in order to reduce network or USB traffic. This parameter does not affect value change callbacks Note: This function must be called after `yInitAPI`.

Parameters :

cacheValidityMs an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds.

YAPI.SetDelegate() YAPI.SetDelegate()

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

m +(void) **SetDelegate** :(id) **object**

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

Parameters :

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

YAPI.SetDeviceListValidity() YAPI

YAPI.SetDeviceListValidity()

Modifies the delay between each forced enumeration of the used YoctoHubs.

cpp	static void **ySetDeviceListValidity**(int **deviceListValidity**)
m	+(void) **SetDeviceListValidity** : (int) **deviceListValidity**
pas	**ySetDeviceListValidity**(**deviceListValidity**: LongInt)
vb	procedure **ySetDeviceListValidity**(ByVal **deviceListValidity** As Integer)
cs	void **SetDeviceListValidity**(int **deviceListValidity**)
java	void **SetDeviceListValidity**(int **deviceListValidity**)
uwp	async Task **SetDeviceListValidity**(int **deviceListValidity**)
py	**SetDeviceListValidity**(**deviceListValidity**)
php	function **ySetDeviceListValidity**(**$deviceListValidity**)
es	async **SetDeviceListValidity**(**deviceListValidity**)

By default, the library performs a full enumeration every 10 seconds. To reduce network traffic, you can increase this delay. It's particularly useful when a YoctoHub is connected to the GSM network where traffic is billed. This parameter doesn't impact modules connected by USB, nor the working of module arrival/removal callbacks. Note: you must call this function after `yInitAPI`.

Parameters :

deviceListValidity nubmer of seconds between each enumeration.

YAPI.SetHTTPCallbackCacheDir() YAPI.SetHTTPCallbackCacheDir() YAPI

Enables the HTTP callback cache.

```
php function ySetHTTPCallbackCacheDir( $str_directory)
```

When enabled, this cache reduces the quantity of data sent to the PHP script by 50% to 70%. To enable this cache, the method `ySetHTTPCallbackCacheDir()` must be called before any call to `yRegisterHub()`. This method takes in parameter the path of the directory used for saving data between each callback. This folder must exist and the PHP script needs to have write access to it. It is recommended to use a folder that is not published on the Web server since the library will save some data of Yoctopuce devices into this folder.

Note: This feature is supported by YoctoHub and VirtualHub since version 27750.

Parameters :

str_directory the path of the folder that will be used as cache.

Returns :

nothing.

On failure, throws an exception.

YAPI.SetNetworkTimeout() YAPI

YAPI.SetNetworkTimeout()

Modifies the network connection delay for `yRegisterHub()` and `yUpdateDeviceList()`.

cpp	static void **ySetNetworkTimeout**(int **networkMsTimeout**)
m	+(void) **SetNetworkTimeout** : (int) **networkMsTimeout**
pas	**ySetNetworkTimeout**(**networkMsTimeout**: LongInt)
vb	procedure **ySetNetworkTimeout**(ByVal **networkMsTimeout** As Integer)
cs	void **SetNetworkTimeout**(int **networkMsTimeout**)
java	void **SetNetworkTimeout**(int **networkMsTimeout**)
uwp	async Task **SetNetworkTimeout**(int **networkMsTimeout**)
py	**SetNetworkTimeout**(**networkMsTimeout**)
php	function **ySetNetworkTimeout**(**$networkMsTimeout**)
es	async **SetNetworkTimeout**(**networkMsTimeout**)
dnp	static void **SetNetworkTimeout**(int **networkMsTimeout**)
cp	static void **SetNetworkTimeout**(int **networkMsTimeout**)

This delay impacts only the YoctoHubs and VirtualHub which are accessible through the network. By default, this delay is of 20000 milliseconds, but depending or you network you may want to change this delay, gor example if your network infrastructure is based on a GSM connection.

Parameters :

networkMsTimeout the network connection delay in milliseconds.

YAPI.SetTimeout() YAPI

YAPI.SetTimeout()

Invoke the specified callback function after a given timeout.

```
js  function ySetTimeout( callback, ms_timeout, args)
es  SetTimeout( callback, ms_timeout, args)
```

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

Parameters :

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

YAPI.SetUSBPacketAckMs() YAPI

YAPI.SetUSBPacketAckMs()

Enables the acknowledge of every USB packet received by the Yoctopuce library.

```
java void SetUSBPacketAckMs( int pktAckDelay)
```

This function allows the library to run on Android phones that tend to loose USB packets. By default, this feature is disabled because it doubles the number of packets sent and slows down the API considerably. Therefore, the acknowledge of incoming USB packets should only be enabled on phones or tablets that loose USB packets. A delay of 50 milliseconds is generally enough. In case of doubt, contact Yoctopuce support. To disable USB packets acknowledge, call this function with the value 0. Note: this feature is only available on Android.

Parameters :

pktAckDelay then number of milliseconds before the module

YAPI.Sleep() YAPI

YAPI.Sleep()

Pauses the execution flow for a specified duration.

```
js    function ySleep( ms_duration, errmsg)
cpp   YRETCODE ySleep( unsigned ms_duration, string errmsg)
m     +(YRETCODE) Sleep :(unsigned) ms_duration :(NSError **) errmsg
pas   integer ySleep( ms_duration: integer, var errmsg: string): integer
vb    function ySleep( ByVal ms_duration As Integer,
                       ByRef errmsg As String) As Integer
cs    static int Sleep( int ms_duration, ref string errmsg)
java  int Sleep( long ms_duration)
uwp   async Task<int> Sleep( ulong ms_duration)
py    Sleep( ms_duration, errmsg=None)
php   function ySleep( $ms_duration, &$errmsg)
es    async Sleep( ms_duration, errmsg)
```

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

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

Parameters :

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

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

YAPI.TestHub() YAPI
YAPI.TestHub()

Test if the hub is reachable.

```
cpp   YRETCODE yTestHub( string url, int mstimeout, string errmsg)
m     +(YRETCODE) TestHub : (NSString*) url
                          : (int) mstimeout
                          : (NSError**) errmsg
pas   integer yTestHub( url: string,
                        mstimeout: integer,
                        var errmsg: string): integer
vb    function yTestHub( ByVal url As String,
                         ByVal mstimeout As Integer,
                         ByRef errmsg As String) As Integer
cs    static int TestHub( string url, int mstimeout, ref string errmsg)
java  int TestHub( String url, int mstimeout)
uwp   async Task<int> TestHub( string url, uint mstimeout)
py    TestHub( url, mstimeout, errmsg=None)
php   function yTestHub( $url, $mstimeout, &$errmsg)
es    async TestHub( url, mstimeout, errmsg)
dnp   static string TestHub( string url, int mstimeout)
cp    static string TestHub( string, int mstimeout)
```

This method do not register the hub, it only test if the hub is usable. The url parameter follow the same convention as the `yRegisterHub` method. This method is useful to verify the authentication parameters for a hub. It is possible to force this method to return after mstimeout milliseconds.

Parameters :

url	a string containing either **"usb"**,**"callback"** or the root URL of the hub to monitor
mstimeout	the number of millisecond available to test the connection.
errmsg	a string passed by reference to receive any error message.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure returns a negative error code.

YAPI.TriggerHubDiscovery() YAPI

YAPI.TriggerHubDiscovery()

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

cpp	YRETCODE **yTriggerHubDiscovery**(string **errmsg**)
m	+(YRETCODE) **TriggerHubDiscovery** : (NSError**) **errmsg**
pas	integer **yTriggerHubDiscovery**(var **errmsg**: string): integer
vb	function **yTriggerHubDiscovery**(ByRef **errmsg** As String) As Integer
cs	static int **TriggerHubDiscovery**(ref string **errmsg**)
java	int **TriggerHubDiscovery**( )
uwp	Task<int> **TriggerHubDiscovery**( )
py	**TriggerHubDiscovery**(**errmsg**=None)
es	async **TriggerHubDiscovery**(**errmsg**)

Parameters :

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

Returns :

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

YAPI.UnregisterHub() YAPI

YAPI.UnregisterHub()

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

```
js    function yUnregisterHub( url)
cpp   void yUnregisterHub( string url)
m     +(void) UnregisterHub :(NSString *) url
pas   yUnregisterHub( url: string)
vb    procedure yUnregisterHub( ByVal url As String)
cs    static void UnregisterHub( string url)
java  void UnregisterHub( String url)
uwp   async Task UnregisterHub( string url)
py    UnregisterHub( url)
php   function yUnregisterHub( $url)
es    async UnregisterHub( url)
```

Parameters :

url a string containing either **"usb"** or the

YAPI.UpdateDeviceList() YAPI

YAPI.UpdateDeviceList()

Triggers a (re)detection of connected Yoctopuce modules.

```
js    function yUpdateDeviceList( errmsg)
cpp   YRETCODE yUpdateDeviceList( string errmsg)
m     +(YRETCODE) UpdateDeviceList :(NSError**) errmsg
pas   integer yUpdateDeviceList( var errmsg: string): integer
vb    function yUpdateDeviceList( ByRef errmsg As String) As YRETCODE
cs    static YRETCODE UpdateDeviceList( ref string errmsg)
java  int UpdateDeviceList( )
uwp   async Task<int> UpdateDeviceList( )
py    UpdateDeviceList( errmsg=None)
php   function yUpdateDeviceList( &$errmsg)
es    async UpdateDeviceList( errmsg)
```

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

This function can be called as frequently as desired to refresh the device list and to make the application aware of hot-plug events. However, since device detection is quite a heavy process, UpdateDeviceList shouldn't be called more than once every two seconds.

Parameters :

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

Returns :

`YAPI_SUCCESS` when the call succeeds.

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

YAPI.UpdateDeviceList_async() YAPI

YAPI.UpdateDeviceList_async()

Triggers a (re)detection of connected Yoctopuce modules.

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

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

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

24.2. Class YModule

Global parameters control interface for all Yoctopuce devices

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

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

js	<script type='text/javascript' src='yocto_api.js'></script>
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
uwp	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *
php	require_once('yocto_api.php');
es	in HTML: <script src="../../lib/yocto_api.js"></script> in node.js: require('yoctolib-es2017/yocto_api.js');
dnp	import YoctoProxyAPI.YModuleProxy
cp	#include "yocto_module_proxy.h"
vi	YModule.vi
ml	import YoctoProxyAPI.YModuleProxy"

Global functions
YModule.FindModule(**func**) Allows you to find a module from its serial number or from its logical name.
YModule.FindModuleInContext(**yctx**, **func**) Retrieves a module for a given identifier in a YAPI context.
YModule.FirstModule() Starts the enumeration of modules currently accessible.
`YModule` properties
module→Beacon *[writable]* State of the localization beacon.
module→FirmwareRelease *[read-only]* Version of the firmware embedded in the module.
module→FunctionId *[read-only]* Retrieves the hardware identifier of the *n*th function on the module.
module→HardwareId *[read-only]* Unique hardware identifier of the module.
module→IsOnline *[read-only]* Checks if the module is currently reachable.
module→LogicalName *[writable]* Logical name of the module.
module→Luminosity *[writable]* Luminosity of the module informative LEDs (from 0 to 100).

module→ProductId *[read-only]*

USB device identifier of the module.

module→ProductName *[read-only]*

Commercial name of the module, as set by the factory.

module→ProductRelease *[read-only]*

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

module→SerialNumber *[read-only]*

Serial number of the module, as set by the factory.

`YModule` methods

module→checkFirmware(path, onlynew)

Tests whether the byn file is valid for this module.

module→clearCache()

Invalidates the cache.

module→describe()

Returns a descriptive text that identifies the module.

module→download(pathname)

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

module→functionBaseType(functionIndex)

Retrieves the base type of the *n*th function on the module.

module→functionCount()

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

module→functionId(functionIndex)

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

module→functionName(functionIndex)

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

module→functionType(functionIndex)

Retrieves the type of the *n*th function on the module.

module→functionValue(functionIndex)

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

module→get_allSettings()

Returns all the settings and uploaded files of the module.

module→get_beacon()

Returns the state of the localization beacon.

module→get_errorMessage()

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

module→get_errorType()

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

module→get_firmwareRelease()

Returns the version of the firmware embedded in the module.

module→get_functionIds(funType)

Retrieve all hardware identifier that match the type passed in argument.

module→get_hardwareId()

Returns the unique hardware identifier of the module.

module→get_icon2d()

Returns the icon of the module.

module→get_lastLogs()

Returns a string with last logs of the module.

module→get_logicalName()

Returns the logical name of the module.

module→get_luminosity()

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

module→get_parentHub()

Returns the serial number of the YoctoHub on which this module is connected.

module→get_persistentSettings()

Returns the current state of persistent module settings.

module→get_productId()

Returns the USB device identifier of the module.

module→get_productName()

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

module→get_productRelease()

Returns the release number of the module hardware, preprogrammed at the factory.

module→get_rebootCountdown()

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

module→get_serialNumber()

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

module→get_subDevices()

Returns a list of all the modules that are plugged into the current module.

module→get_upTime()

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

module→get_url()

Returns the URL used to access the module.

module→get_usbCurrent()

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

module→get_userData()

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

module→get_userVar()

Returns the value previously stored in this attribute.

module→hasFunction(funcId)

Tests if the device includes a specific function.

module→isOnline()

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

module→isOnline_async(callback, context)

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

module→load(msValidity)

Preloads the module cache with a specified validity duration.

module→load_async(msValidity, callback, context)

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

module→log(text)

Adds a text message to the device logs.

module→nextModule()

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

module→reboot(secBeforeReboot)

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

module→registerBeaconCallback(callback)

Register a callback function, to be called when the localization beacon of the module has been changed.

module→registerConfigChangeCallback(callback)

Register a callback function, to be called when a persistent settings in a device configuration has been changed (e.g.

module→registerLogCallback(callback)

Registers a device log callback function.

module→revertFromFlash()

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

module→saveToFlash()

Saves current settings in the nonvolatile memory of the module.

module→set_allSettings(settings)

Restores all the settings of the device.

module→set_allSettingsAndFiles(settings)

Restores all the settings and uploaded files to the module.

module→set_beacon(newval)

Turns on or off the module localization beacon.

module→set_logicalName(newval)

Changes the logical name of the module.

module→set_luminosity(newval)

Changes the luminosity of the module informative leds.

module→set_userData(data)

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

module→set_userVar(newval)

Stores a 32 bit value in the device RAM.

module→triggerConfigChangeCallback()

Triggers a configuration change callback, to check if they are supported or not.

module→triggerFirmwareUpdate(secBeforeReboot)

Schedules a module reboot into special firmware update mode.

module→updateFirmware(path)

Prepares a firmware update of the module.

module→updateFirmwareEx(path, force)

Prepares a firmware update of the module.

module→wait_async(callback, context)

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

YModule.FindModule() YModule
YModule.FindModule()

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

```
js    function yFindModule( func)
cpp   YModule* yFindModule( string func)
m     +(YModule*) FindModule : (NSString*) func
pas   TYModule yFindModule( func: string): TYModule
vb    function yFindModule( ByVal func As String) As YModule
cs    static YModule FindModule( string func)
java  static YModule FindModule( String func)
uwp   static YModule FindModule( string func)
py    FindModule( func)
php   function yFindModule( $func)
es    static FindModule( func)
dnp   static YModuleProxy FindModule( string func)
cp    static YModuleProxy * FindModule( string func)
```

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

If a call to this object's is_online() method returns FALSE although you are certain that the device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

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

Returns :

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

YModule.FindModuleInContext() YModule
YModule.FindModuleInContext()

Retrieves a module for a given identifier in a YAPI context.

```
java  static YModule FindModuleInContext( YAPIContext yctx, String func)
uwp   static YModule FindModuleInContext( YAPIContext yctx, string func)
es    static FindModuleInContext( yctx, func)
```

The identifier can be specified using several formats:

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

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

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the module, for instance `MyDevice.module`.

Returns :

a `YModule` object allowing you to drive the module.

YModule.FirstModule() YModule

YModule.FirstModule()

Starts the enumeration of modules currently accessible.

js	function **yFirstModule**( )
cpp	YModule * **yFirstModule**( )
m	+(YModule*) **FirstModule**
pas	TYModule **yFirstModule**( ): TYModule
vb	function **yFirstModule**( ) As YModule
cs	static YModule **FirstModule**( )
java	static YModule **FirstModule**( )
uwp	static YModule **FirstModule**( )
py	**FirstModule**( )
php	function **yFirstModule**( )
es	static **FirstModule**( )

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

Returns :

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

module→Beacon YModule

State of the localization beacon.

dnp int **Beacon**

Possible values:

```
Y_BEACON_INVALID = 0
Y_BEACON_OFF     = 1
Y_BEACON_ON      = 2
```

Writable. Turns on or off the module localization beacon.

module→FirmwareRelease YModule

Version of the firmware embedded in the module.

dnp string **FirmwareRelease**

module→FunctionId YModule

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

dnp string **FunctionId**

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

module→HardwareId YModule

Unique hardware identifier of the module.

dnp string **HardwareId**

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

module→IsOnline YModule

Checks if the module is currently reachable.

dnp bool **IsOnline**

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

module→LogicalName YModule

Logical name of the module.

dnp string **LogicalName**

Writable. You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

module→Luminosity YModule

Luminosity of the module informative LEDs (from 0 to 100).

dnp int **Luminosity**

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

module→ProductId YModule

USB device identifier of the module.

dnp int **ProductId**

module→ProductName YModule

Commercial name of the module, as set by the factory.

dnp string **ProductName**

module→ProductRelease YModule

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

dnp int **ProductRelease**

The original hardware release returns value 1, revision B returns value 2, etc.

module→SerialNumber YModule

Serial number of the module, as set by the factory.

dnp string **SerialNumber**

module→checkFirmware() YModule

Tests whether the byn file is valid for this module.

```
js    function checkFirmware( path, onlynew)
cpp   string checkFirmware( string path, bool onlynew)
m     -(NSString*) checkFirmware : (NSString*) path
                                 : (bool) onlynew
pas   string checkFirmware( path: string, onlynew: boolean): string
vb    function checkFirmware( ByVal path As String, ByVal onlynew As Boolean) As String
cs    string checkFirmware( string path, bool onlynew)
java  String checkFirmware( String path, boolean onlynew)
uwp   async Task<string> checkFirmware( string path, bool onlynew)
py    checkFirmware( path, onlynew)
php   function checkFirmware( $path, $onlynew)
es    async checkFirmware( path, onlynew)
dnp   string checkFirmware( string path, bool onlynew)
cp    string checkFirmware( string path, bool onlynew)
cmd   YModule target checkFirmware path onlynew
```

This method is useful to test if the module needs to be updated. It is possible to pass a directory as argument instead of a file. In this case, this method returns the path of the most recent appropriate `.byn` file. If the parameter `onlynew` is true, the function discards firmwares that are older or equal to the installed firmware.

Parameters :

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

onlynew returns only files that are strictly newer

Returns :

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

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

module→clearCache() YModule

Invalidates the cache.

js	function **clearCache**( )
cpp	void **clearCache**( )
m	-(void) **clearCache**
pas	**clearCache**( )
vb	procedure **clearCache**( )
cs	void **clearCache**( )
java	void **clearCache**( )
py	**clearCache**( )
php	function **clearCache**( )
es	async **clearCache**( )

Invalidates the cache of the module attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

module→describe() YModule

Returns a descriptive text that identifies the module.

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

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

Returns :
a string that describes the module

module→download() YModule

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

```
js    function download( pathname)
cpp   string download( string pathname)
m     -(NSMutableData*) download : (NSString*) pathname
pas   TByteArray download( pathname: string): TByteArray
vb    function download( ByVal pathname As String) As Byte
cs    byte[] download( string pathname)
java  byte[] download( String pathname)
uwp   async Task<byte[]> download( string pathname)
py    download( pathname)
php   function download( $pathname)
es    async download( pathname)
dnp   byte[] download( string pathname)
cp    string download( string pathname)
cmd   YModule target download pathname
```

Parameters :

pathname name of the new file to load

Returns :

a binary buffer with the file content

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

module→functionBaseType() YModule

Retrieves the base type of the *n*th function on the module.

```
js    function functionBaseType( functionIndex)
cpp   string functionBaseType( int functionIndex)
pas   string functionBaseType( functionIndex: integer): string
vb    function functionBaseType( ByVal functionIndex As Integer) As String
cs    string functionBaseType( int functionIndex)
java  String functionBaseType( int functionIndex)
py    functionBaseType( functionIndex)
php   function functionBaseType( $functionIndex)
es    async functionBaseType( functionIndex)
```

For instance, the base type of all measuring functions is "Sensor".

Parameters :

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

Returns :

a string corresponding to the base type of the function

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

module→functionCount() YModule

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

js	function **functionCount**( )
cpp	int **functionCount**( )
m	-(int) **functionCount**
pas	integer **functionCount**( ): integer
vb	function **functionCount**( ) As Integer
cs	int **functionCount**( )
java	int **functionCount**( )
py	**functionCount**( )
php	function **functionCount**( )
es	async **functionCount**( )

Returns :
the number of functions on the module

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

module→functionId() YModule

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

```
js    function functionId( functionIndex)
cpp   string functionId( int functionIndex)
m     -(NSString*) functionId : (int) functionIndex
pas   string functionId( functionIndex: integer): string
vb    function functionId( ByVal functionIndex As Integer) As String
cs    string functionId( int functionIndex)
java  String functionId( int functionIndex)
py    functionId( functionIndex)
php   function functionId( $functionIndex)
es    async functionId( functionIndex)
```

Parameters :

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

Returns :

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

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

module→functionName() YModule

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

```
js      function functionName( functionIndex)
cpp     string functionName( int functionIndex)
m       -(NSString*) functionName : (int) functionIndex
pas     string functionName( functionIndex: integer): string
vb      function functionName( ByVal functionIndex As Integer) As String
cs      string functionName( int functionIndex)
java    String functionName( int functionIndex)
py      functionName( functionIndex)
php     function functionName( $functionIndex)
es      async functionName( functionIndex)
```

Parameters :

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

Returns :

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

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

module→functionType() YModule

Retrieves the type of the *n*th function on the module.

```
js    function functionType( functionIndex)
cpp   string functionType( int functionIndex)
pas   string functionType( functionIndex: integer): string
vb    function functionType( ByVal functionIndex As Integer) As String
cs    string functionType( int functionIndex)
java  String functionType( int functionIndex)
py    functionType( functionIndex)
php   function functionType( $functionIndex)
es    async functionType( functionIndex)
```

Parameters :

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

Returns :

a string corresponding to the type of the function

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

module→functionValue() YModule

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

```
js    function functionValue( functionIndex)
cpp   string functionValue( int functionIndex)
m     -(NSString*) functionValue : (int) functionIndex
pas   string functionValue( functionIndex: integer): string
vb    function functionValue( ByVal functionIndex As Integer) As String
cs    string functionValue( int functionIndex)
java  String functionValue( int functionIndex)
py    functionValue( functionIndex)
php   function functionValue( $functionIndex)
es    async functionValue( functionIndex)
```

Parameters :

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

Returns :

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

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

module→get_allSettings() module→allSettings() YModule

Returns all the settings and uploaded files of the module.

```
js      function get_allSettings( )
cpp     string get_allSettings( )
m       -(NSMutableData*) allSettings
pas     TByteArray get_allSettings( ): TByteArray
vb      function get_allSettings( ) As Byte
cs      byte[] get_allSettings( )
java    byte[] get_allSettings( )
uwp     async Task<byte[]> get_allSettings( )
py      get_allSettings( )
php     function get_allSettings( )
es      async get_allSettings( )
dnp     byte[] get_allSettings( )
cp      string get_allSettings( )
cmd     YModule target get_allSettings
```

Useful to backup all the logical names, calibrations parameters, and uploaded files of a device.

Returns :
a binary buffer with all the settings.

On failure, throws an exception or returns an binary object of size 0.

module→get_beacon() YModule
module→beacon()

Returns the state of the localization beacon.

```
js    function get_beacon( )
cpp   Y_BEACON_enum get_beacon( )
m     -(Y_BEACON_enum) beacon
pas   Integer get_beacon( ): Integer
vb    function get_beacon( ) As Integer
cs    int get_beacon( )
java  int get_beacon( )
uwp   async Task<int> get_beacon( )
py    get_beacon( )
php   function get_beacon( )
es    async get_beacon( )
dnp   int get_beacon( )
cp    int get_beacon( )
cmd   YModule target get_beacon
```

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

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

module→get_errorMessage() YModule
module→errorMessage()

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

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

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

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

module→get_errorType() YModule
module→errorType()

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

```
js    function get_errorType( )
cpp   YRETCODE get_errorType( )
m     -(YRETCODE) errorType
pas   YRETCODE get_errorType( ): YRETCODE
vb    function get_errorType( ) As YRETCODE
cs    YRETCODE get_errorType( )
java  int get_errorType( )
py    get_errorType( )
php   function get_errorType( )
es    get_errorType( )
```

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

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

module→get_firmwareRelease() YModule
module→firmwareRelease()

Returns the version of the firmware embedded in the module.

js	function **get_firmwareRelease**( )
cpp	string **get_firmwareRelease**( )
m	-(NSString*) firmwareRelease
pas	string **get_firmwareRelease**( ): string
vb	function **get_firmwareRelease**( ) As String
cs	string **get_firmwareRelease**( )
java	String **get_firmwareRelease**( )
uwp	async Task<string> **get_firmwareRelease**( )
py	**get_firmwareRelease**( )
php	function **get_firmwareRelease**( )
es	async **get_firmwareRelease**( )
dnp	string **get_firmwareRelease**( )
cp	string **get_firmwareRelease**( )
cmd	YModule **target get_firmwareRelease**

Returns :
a string corresponding to the version of the firmware embedded in the module

On failure, throws an exception or returns `Y_FIRMWARERELEASE_INVALID`.

module→get_functionIds() YModule
module→functionIds()

Retrieve all hardware identifier that match the type passed in argument.

```
js    function get_functionIds( funType)
cpp   vector<string> get_functionIds( string funType)
m     -(NSMutableArray*) functionIds : (NSString*) funType
pas   TStringArray get_functionIds( funType: string): TStringArray
vb    function get_functionIds( ByVal funType As String) As List
cs    List<string> get_functionIds( string funType)
java  ArrayList<String> get_functionIds( String funType)
uwp   async Task<List<string>> get_functionIds( string funType)
py    get_functionIds( funType)
php   function get_functionIds( $funType)
es    async get_functionIds( funType)
dnp   string[] get_functionIds( string funType)
cp    vector<string> get_functionIds( string funType)
cmd   YModule target get_functionIds funType
```

Parameters :

funType The type of function (Relay, LightSensor, Voltage,...)

Returns :

an array of strings.

module→get_hardwareId() module→hardwareId() YModule

Returns the unique hardware identifier of the module.

```
js    function get_hardwareId( )
cpp   string get_hardwareId( )
m     -(NSString*) hardwareId
vb    function get_hardwareId( ) As String
cs    string get_hardwareId( )
java  String get_hardwareId( )
py    get_hardwareId( )
php   function get_hardwareId( )
es    async get_hardwareId( )
dnp   string get_hardwareId( )
cp    string get_hardwareId( )
pas   string get_hardwareId( ): string
uwp   async Task<string> get_hardwareId( )
cmd   YModule target get_hardwareId
```

The unique hardware identifier is made of the device serial number followed by string ".module".

Returns :
a string that uniquely identifies the module

module→get_icon2d() YModule
module→icon2d()

Returns the icon of the module.

js	function **get_icon2d**( )
cpp	string **get_icon2d**( )
m	-(NSMutableData*) icon2d
pas	TByteArray **get_icon2d**( ): TByteArray
vb	function **get_icon2d**( ) As Byte
cs	byte[] **get_icon2d**( )
java	byte[] **get_icon2d**( )
uwp	async Task<byte[]> **get_icon2d**( )
py	**get_icon2d**( )
php	function **get_icon2d**( )
es	async **get_icon2d**( )
dnp	byte[] **get_icon2d**( )
cp	string **get_icon2d**( )
cmd	YModule **target get_icon2d**

The icon is a PNG image and does not exceeds 1536 bytes.

Returns :
a binary buffer with module icon, in png format. On failure, throws an exception or returns `YAPI_INVALID_STRING`.

module→get_lastLogs() YModule
module→lastLogs()

Returns a string with last logs of the module.

js	function **get_lastLogs**( )
cpp	string **get_lastLogs**( )
m	-(NSString*) lastLogs
pas	string **get_lastLogs**( ): string
vb	function **get_lastLogs**( ) As String
cs	string **get_lastLogs**( )
java	String **get_lastLogs**( )
uwp	async Task<string> **get_lastLogs**( )
py	**get_lastLogs**( )
php	function **get_lastLogs**( )
es	async **get_lastLogs**( )
dnp	string **get_lastLogs**( )
cp	string **get_lastLogs**( )
cmd	YModule **target get_lastLogs**

This method return only logs that are still in the module.

Returns :
a string with last logs of the module. On failure, throws an exception or returns `YAPI_INVALID_STRING`.

module→get_logicalName() YModule
module→logicalName()

Returns the logical name of the module.

```
js    function get_logicalName( )
cpp   string get_logicalName( )
m     -(NSString*) logicalName
pas   string get_logicalName( ): string
vb    function get_logicalName( ) As String
cs    string get_logicalName( )
java  String get_logicalName( )
uwp   async Task<string> get_logicalName( )
py    get_logicalName( )
php   function get_logicalName( )
es    async get_logicalName( )
dnp   string get_logicalName( )
cp    string get_logicalName( )
cmd   YModule target get_logicalName
```

Returns :
a string corresponding to the logical name of the module

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

module→get_luminosity() YModule
module→luminosity()

Returns the luminosity of the module informative LEDs (from 0 to 100).

```
js    function get_luminosity( )
cpp   int get_luminosity( )
m     -(int) luminosity
pas   LongInt get_luminosity( ): LongInt
vb    function get_luminosity( ) As Integer
cs    int get_luminosity( )
java  int get_luminosity( )
uwp   async Task<int> get_luminosity( )
py    get_luminosity( )
php   function get_luminosity( )
es    async get_luminosity( )
dnp   int get_luminosity( )
cp    int get_luminosity( )
cmd   YModule target get_luminosity
```

Returns :
an integer corresponding to the luminosity of the module informative LEDs (from 0 to 100)

On failure, throws an exception or returns `Y_LUMINOSITY_INVALID`.

module→get_parentHub() YModule
module→parentHub()

Returns the serial number of the YoctoHub on which this module is connected.

js	function **get_parentHub**( )
cpp	string **get_parentHub**( )
m	-(NSString*) parentHub
pas	string **get_parentHub**( ): string
vb	function **get_parentHub**( ) As String
cs	string **get_parentHub**( )
java	String **get_parentHub**( )
uwp	async Task<string> **get_parentHub**( )
py	**get_parentHub**( )
php	function **get_parentHub**( )
es	async **get_parentHub**( )
dnp	string **get_parentHub**( )
cp	string **get_parentHub**( )
cmd	YModule **target get_parentHub**

If the module is connected by USB, or if the module is the root YoctoHub, an empty string is returned.

Returns :
a string with the serial number of the YoctoHub or an empty string

module→get_persistentSettings() module→persistentSettings()

YModule

Returns the current state of persistent module settings.

js	function **get_persistentSettings**( )
cpp	Y_PERSISTENTSETTINGS_enum **get_persistentSettings**( )
m	-(Y_PERSISTENTSETTINGS_enum) persistentSettings
pas	Integer **get_persistentSettings**( ): Integer
vb	function **get_persistentSettings**( ) As Integer
cs	int **get_persistentSettings**( )
java	int **get_persistentSettings**( )
uwp	async Task<int> **get_persistentSettings**( )
py	**get_persistentSettings**( )
php	function **get_persistentSettings**( )
es	async **get_persistentSettings**( )
dnp	int **get_persistentSettings**( )
cp	int **get_persistentSettings**( )
cmd	YModule **target get_persistentSettings**

Returns :

a value among `Y_PERSISTENTSETTINGS_LOADED`, `Y_PERSISTENTSETTINGS_SAVED` and `Y_PERSISTENTSETTINGS_MODIFIED` corresponding to the current state of persistent module settings

On failure, throws an exception or returns `Y_PERSISTENTSETTINGS_INVALID`.

module→get_productId() YModule
module→productId()

Returns the USB device identifier of the module.

```
js    function get_productId( )
cpp   int get_productId( )
m     -(int) productId
pas   LongInt get_productId( ): LongInt
vb    function get_productId( ) As Integer
cs    int get_productId( )
java  int get_productId( )
uwp   async Task<int> get_productId( )
py    get_productId( )
php   function get_productId( )
es    async get_productId( )
dnp   int get_productId( )
cp    int get_productId( )
cmd   YModule target get_productId
```

Returns :
an integer corresponding to the USB device identifier of the module

On failure, throws an exception or returns `Y_PRODUCTID_INVALID`.

module→get_productName() module→productName() YModule

Returns the commercial name of the module, as set by the factory.

js	function **get_productName**( )
cpp	string **get_productName**( )
m	-(NSString*) productName
pas	string **get_productName**( ): string
vb	function **get_productName**( ) As String
cs	string **get_productName**( )
java	String **get_productName**( )
uwp	async Task<string> **get_productName**( )
py	**get_productName**( )
php	function **get_productName**( )
es	async **get_productName**( )
dnp	string **get_productName**( )
cp	string **get_productName**( )
cmd	YModule **target get_productName**

Returns :
a string corresponding to the commercial name of the module, as set by the factory

On failure, throws an exception or returns `Y_PRODUCTNAME_INVALID`.

module→get_productRelease() YModule
module→productRelease()

Returns the release number of the module hardware, preprogrammed at the factory.

js	function **get_productRelease**( )
cpp	int **get_productRelease**( )
m	-(int) productRelease
pas	LongInt **get_productRelease**( ): LongInt
vb	function **get_productRelease**( ) As Integer
cs	int **get_productRelease**( )
java	int **get_productRelease**( )
uwp	async Task<int> **get_productRelease**( )
py	**get_productRelease**( )
php	function **get_productRelease**( )
es	async **get_productRelease**( )
dnp	int **get_productRelease**( )
cp	int **get_productRelease**( )
cmd	YModule **target get_productRelease**

The original hardware release returns value 1, revision B returns value 2, etc.

Returns :
an integer corresponding to the release number of the module hardware, preprogrammed at the factory

On failure, throws an exception or returns `Y_PRODUCTRELEASE_INVALID`.

module→get_rebootCountdown() YModule
module→rebootCountdown()

Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

js	function **get_rebootCountdown**( )
cpp	int **get_rebootCountdown**( )
m	-(int) rebootCountdown
pas	LongInt **get_rebootCountdown**( ): LongInt
vb	function **get_rebootCountdown**( ) As Integer
cs	int **get_rebootCountdown**( )
java	int **get_rebootCountdown**( )
uwp	async Task<int> **get_rebootCountdown**( )
py	**get_rebootCountdown**( )
php	function **get_rebootCountdown**( )
es	async **get_rebootCountdown**( )
dnp	int **get_rebootCountdown**( )
cp	int **get_rebootCountdown**( )
cmd	YModule **target get_rebootCountdown**

Returns :

an integer corresponding to the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled

On failure, throws an exception or returns `Y_REBOOTCOUNTDOWN_INVALID`.

module→get_serialNumber() YModule
module→serialNumber()

Returns the serial number of the module, as set by the factory.

```
js      function get_serialNumber( )
cpp     string get_serialNumber( )
m       -(NSString*) serialNumber
pas     string get_serialNumber( ): string
vb      function get_serialNumber( ) As String
cs      string get_serialNumber( )
java    String get_serialNumber( )
uwp     async Task<string> get_serialNumber( )
py      get_serialNumber( )
php     function get_serialNumber( )
es      async get_serialNumber( )
dnp     string get_serialNumber( )
cp      string get_serialNumber( )
cmd     YModule target get_serialNumber
```

Returns :
a string corresponding to the serial number of the module, as set by the factory

On failure, throws an exception or returns `Y_SERIALNUMBER_INVALID`.

module→get_subDevices() YModule
module→subDevices()

Returns a list of all the modules that are plugged into the current module.

js	function **get_subDevices**( )
cpp	vector<string> **get_subDevices**( )
m	-(NSMutableArray*) subDevices
pas	TStringArray **get_subDevices**( ): TStringArray
vb	function **get_subDevices**( ) As List
cs	List<string> **get_subDevices**( )
java	ArrayList<String> **get_subDevices**( )
uwp	async Task<List<string>> **get_subDevices**( )
py	**get_subDevices**( )
php	function **get_subDevices**( )
es	async **get_subDevices**( )
dnp	string[] **get_subDevices**( )
cp	vector<string> **get_subDevices**( )
cmd	YModule **target get_subDevices**

This method only makes sense when called for a YoctoHub/VirtualHub. Otherwise, an empty array will be returned.

Returns :
an array of strings containing the sub modules.

module→get_upTime() YModule
module→upTime()

Returns the number of milliseconds spent since the module was powered on.

```
js    function get_upTime( )
cpp   s64 get_upTime( )
m     -(s64) upTime
pas   int64 get_upTime( ): int64
vb    function get_upTime( ) As Long
cs    long get_upTime( )
java  long get_upTime( )
uwp   async Task<long> get_upTime( )
py    get_upTime( )
php   function get_upTime( )
es    async get_upTime( )
dnp   long get_upTime( )
cp    s64 get_upTime( )
cmd   YModule target get_upTime
```

Returns :
an integer corresponding to the number of milliseconds spent since the module was powered on

On failure, throws an exception or returns `Y_UPTIME_INVALID`.

module→get_url() YModule
module→url()

Returns the URL used to access the module.

```
js      function get_url( )
cpp     string get_url( )
m       -(NSString*) url
pas     string get_url( ): string
vb      function get_url( ) As String
cs      string get_url( )
java    String get_url( )
uwp     async Task<string> get_url( )
py      get_url( )
php     function get_url( )
es      async get_url( )
dnp     string get_url( )
cp      string get_url( )
cmd     YModule target get_url
```

If the module is connected by USB, the string 'usb' is returned.

Returns :
a string with the URL of the module.

module→get_usbCurrent() YModule
module→usbCurrent()

Returns the current consumed by the module on the USB bus, in milli-amps.

js	function **get_usbCurrent**( )
cpp	int **get_usbCurrent**( )
m	-(int) usbCurrent
pas	LongInt **get_usbCurrent**( ): LongInt
vb	function **get_usbCurrent**( ) As Integer
cs	int **get_usbCurrent**( )
java	int **get_usbCurrent**( )
uwp	async Task<int> **get_usbCurrent**( )
py	**get_usbCurrent**( )
php	function **get_usbCurrent**( )
es	async **get_usbCurrent**( )
dnp	int **get_usbCurrent**( )
cp	int **get_usbCurrent**( )
cmd	YModule **target get_usbCurrent**

Returns :
an integer corresponding to the current consumed by the module on the USB bus, in milli-amps

On failure, throws an exception or returns `Y_USBCURRENT_INVALID`.

module→get_userData() YModule
module→userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
js      function get_userData( )
cpp     void * get_userData( )
m       -(id) userData
pas     Tobject get_userData( ): Tobject
vb      function get_userData( ) As Object
cs      object get_userData( )
java    Object get_userData( )
py      get_userData( )
php     function get_userData( )
es      async get_userData( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

module→get_userVar() YModule
module→userVar()

Returns the value previously stored in this attribute.

```
js    function get_userVar( )
cpp   int get_userVar( )
m     -(int) userVar
pas   LongInt get_userVar( ): LongInt
vb    function get_userVar( ) As Integer
cs    int get_userVar( )
java  int get_userVar( )
uwp   async Task<int> get_userVar( )
py    get_userVar( )
php   function get_userVar( )
es    async get_userVar( )
dnp   int get_userVar( )
cp    int get_userVar( )
cmd   YModule target get_userVar
```

On startup and after a device reboot, the value is always reset to zero.

Returns :
an integer corresponding to the value previously stored in this attribute

On failure, throws an exception or returns `Y_USERVAR_INVALID`.

module→hasFunction() YModule

Tests if the device includes a specific function.

```
js    function hasFunction( funcId)
cpp   bool hasFunction( string funcId)
m     -(bool) hasFunction : (NSString*) funcId
pas   boolean hasFunction( funcId: string): boolean
vb    function hasFunction( ByVal funcId As String) As Boolean
cs    bool hasFunction( string funcId)
java  boolean hasFunction( String funcId)
uwp   async Task<bool> hasFunction( string funcId)
py    hasFunction( funcId)
php   function hasFunction( $funcId)
es    async hasFunction( funcId)
dnp   bool hasFunction( string funcId)
cp    bool hasFunction( string funcId)
cmd   YModule target hasFunction funcId
```

This method takes a function identifier and returns a boolean.

Parameters :

funcId the requested function identifier

Returns :

true if the device has the function identifier

module→isOnline() YModule

Checks if the module is currently reachable, without raising any error.

js	function **isOnline**( )
cpp	bool **isOnline**( )
m	-(BOOL) **isOnline**
pas	boolean **isOnline**( ): boolean
vb	function **isOnline**( ) As Boolean
cs	bool **isOnline**( )
java	boolean **isOnline**( )
py	**isOnline**( )
php	function **isOnline**( )
es	async **isOnline**( )
dnp	bool **isOnline**( )
cp	bool **isOnline**( )

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

Returns :
`true` if the module can be reached, and `false` otherwise

module→isOnline_async() YModule

Checks if the module is currently reachable, without raising any error.

```
js function isOnline_async( callback, context)
```

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking Firefox JavaScript VM that does not implement context switching during blocking I/O calls.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving module object and the boolean result

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

module→load() YModule

Preloads the module cache with a specified validity duration.

```
js    function load( msValidity)
cpp   YRETCODE load( int msValidity)
m     -(YRETCODE) load : (u64) msValidity
pas   YRETCODE load( msValidity: u64): YRETCODE
vb    function load( ByVal msValidity As Long) As YRETCODE
cs    YRETCODE load( ulong msValidity)
java  int load( long msValidity)
py    load( msValidity)
php   function load( $msValidity)
es    async load( msValidity)
```

By default, whenever accessing a device, all module attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded module parameters, in milliseconds

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→load_async() YModule

Preloads the module cache with a specified validity duration (asynchronous version).

```
js function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all module attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking Firefox JavaScript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous JavaScript calls for more details.

Parameters :

msValidity an integer corresponding to the validity of the loaded module parameters, in milliseconds

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving module object and the error code (or `YAPI_SUCCESS`)

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

module→log() YModule

Adds a text message to the device logs.

```
js    function log( text)
cpp   int log( string text)
m     -(int) log : (NSString*) text
pas   LongInt log( text: string): LongInt
vb    function log( ByVal text As String) As Integer
cs    int log( string text)
java  int log( String text)
uwp   async Task<int> log( string text)
py    log( text)
php   function log( $text)
es    async log( text)
dnp   int log( string text)
cp    int log( string text)
cmd   YModule target log text
```

This function is useful in particular to trace the execution of HTTP callbacks. If a newline is desired after the message, it must be included in the string.

Parameters :

text the string to append to the logs.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→nextModule() YModule

Continues the module enumeration started using `yFirstModule()`.

js	function **nextModule**( )
cpp	YModule * **nextModule**( )
m	-(YModule*) **nextModule**
pas	TYModule **nextModule**( ): TYModule
vb	function **nextModule**( ) As YModule
cs	YModule **nextModule**( )
java	YModule **nextModule**( )
uwp	YModule **nextModule**( )
py	**nextModule**( )
php	function **nextModule**( )
es	**nextModule**( )

Caution: You can't make any assumption about the returned modules order. If you want to find a specific module, use `Module.findModule()` and a hardwareID or a logical name.

Returns :

a pointer to a `YModule` object, corresponding to the next module found, or a `null` pointer if there are no more modules to enumerate.

module→reboot() YModule

Schedules a simple module reboot after the given number of seconds.

```
js    function reboot( secBeforeReboot)
cpp   int reboot( int secBeforeReboot)
m     -(int) reboot : (int) secBeforeReboot
pas   LongInt reboot( secBeforeReboot: LongInt): LongInt
vb    function reboot( ByVal secBeforeReboot As Integer) As Integer
cs    int reboot( int secBeforeReboot)
java  int reboot( int secBeforeReboot)
uwp   async Task<int> reboot( int secBeforeReboot)
py    reboot( secBeforeReboot)
php   function reboot( $secBeforeReboot)
es    async reboot( secBeforeReboot)
dnp   int reboot( int secBeforeReboot)
cp    int reboot( int secBeforeReboot)
cmd   YModule target reboot secBeforeReboot
```

Parameters :

secBeforeReboot number of seconds before rebooting

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→registerBeaconCallback() YModule

Register a callback function, to be called when the localization beacon of the module has been changed.

```
js    function registerBeaconCallback( callback)
cpp   int registerBeaconCallback( YModuleBeaconCallback callback)
m     -(int) registerBeaconCallback : (YModuleBeaconCallback) callback
pas   LongInt registerBeaconCallback( callback: TYModuleBeaconCallback): LongInt
vb    function registerBeaconCallback( ByVal callback As YModuleBeaconCallback) As Integer
cs    int registerBeaconCallback( BeaconCallback callback)
java  int registerBeaconCallback( BeaconCallback callback)
uwp   async Task<int> registerBeaconCallback( BeaconCallback callback)
py    registerBeaconCallback( callback)
php   function registerBeaconCallback( $callback)
es    async registerBeaconCallback( callback)
```

The callback function should take two arguments: the YModule object of which the beacon has changed, and an integer describing the new beacon state.

Parameters :

callback The callback function to call, or `null` to unregister a

module→registerConfigChangeCallback() YModule

Register a callback function, to be called when a persistent settings in a device configuration has been changed (e.g.

```
js    function registerConfigChangeCallback( callback)
cpp   int registerConfigChangeCallback( YModuleConfigChangeCallback callback)
m     -(int) registerConfigChangeCallback : (YModuleConfigChangeCallback) callback
pas   LongInt registerConfigChangeCallback( callback: TYModuleConfigChangeCallback): LongInt
vb    function registerConfigChangeCallback( ByVal callback As YModuleConfigChangeCallback) As
      Integer
cs    int registerConfigChangeCallback( ConfigChangeCallback callback)
java  int registerConfigChangeCallback( ConfigChangeCallback callback)
uwp   async Task<int> registerConfigChangeCallback( ConfigChangeCallback callback)
py    registerConfigChangeCallback( callback)
php   function registerConfigChangeCallback( $callback)
es    async registerConfigChangeCallback( callback)
```

change of unit, etc).

Parameters :

callback a procedure taking a YModule parameter, or `null`

module→registerLogCallback() YModule

Registers a device log callback function.

js	function **registerLogCallback**(**callback**)
cpp	int **registerLogCallback**(YModuleLogCallback **callback**)
m	-(int) **registerLogCallback** : (YModuleLogCallback) **callback**
pas	LongInt **registerLogCallback**(**callback**: TYModuleLogCallback): LongInt
vb	function **registerLogCallback**(ByVal **callback** As YModuleLogCallback) As Integer
cs	int **registerLogCallback**(LogCallback **callback**)
java	int **registerLogCallback**(LogCallback **callback**)
uwp	async Task<int> **registerLogCallback**(LogCallback **callback**)
py	**registerLogCallback**(**callback**)
php	function **registerLogCallback**(**$callback**)
es	async **registerLogCallback**(**callback**)

This callback will be called each time that a module sends a new log message. Mostly useful to debug a Yoctopuce module.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the module object that emitted the log message, and the character string containing the log.

module→revertFromFlash() YModule

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

js	function **revertFromFlash**( )
cpp	int **revertFromFlash**( )
m	-(int) **revertFromFlash**
pas	LongInt **revertFromFlash**( ): LongInt
vb	function **revertFromFlash**( ) As Integer
cs	int **revertFromFlash**( )
java	int **revertFromFlash**( )
uwp	async Task<int> **revertFromFlash**( )
py	**revertFromFlash**( )
php	function **revertFromFlash**( )
es	async **revertFromFlash**( )
dnp	int **revertFromFlash**( )
cp	int **revertFromFlash**( )
cmd	YModule **target revertFromFlash**

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→saveToFlash() YModule

Saves current settings in the nonvolatile memory of the module.

js	function **saveToFlash**( )
cpp	int **saveToFlash**( )
m	-(int) **saveToFlash**
pas	LongInt **saveToFlash**( ): LongInt
vb	function **saveToFlash**( ) As Integer
cs	int **saveToFlash**( )
java	int **saveToFlash**( )
uwp	async Task<int> **saveToFlash**( )
py	**saveToFlash**( )
php	function **saveToFlash**( )
es	async **saveToFlash**( )
dnp	int **saveToFlash**( )
cp	int **saveToFlash**( )
cmd	YModule **target saveToFlash**

Warning: the number of allowed save operations during a module life is limited (about 100000 cycles). Do not call this function within a loop.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_allSettings() YModule
module→setAllSettings()

Restores all the settings of the device.

```
js      function set_allSettings( settings)
cpp     int set_allSettings( string settings)
m       -(int) setAllSettings : (NSData*) settings
pas     LongInt set_allSettings( settings: TByteArray): LongInt
vb      procedure set_allSettings( ByVal settings As Byte()
cs      int set_allSettings( byte[] settings)
java    int set_allSettings( byte[] settings)
uwp     async Task<int> set_allSettings( byte[] settings)
py      set_allSettings( settings)
php     function set_allSettings( $settings)
es      async set_allSettings( settings)
dnp     int set_allSettings( byte[] settings)
cp      int set_allSettings( string settings)
cmd     YModule target set_allSettings settings
```

Useful to restore all the logical names and calibrations parameters of a module from a backup.Remember to call the `saveToFlash()` method of the module if the modifications must be kept.

Parameters :

settings a binary buffer with all the settings.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_allSettingsAndFiles() YModule
module→setAllSettingsAndFiles()

Restores all the settings and uploaded files to the module.

```
js    function set_allSettingsAndFiles( settings)
cpp   int set_allSettingsAndFiles( string settings)
m     -(int) setAllSettingsAndFiles : (NSData*) settings
pas   LongInt set_allSettingsAndFiles( settings: TByteArray): LongInt
vb    procedure set_allSettingsAndFiles( ByVal settings As Byte()
cs    int set_allSettingsAndFiles( byte[] settings)
java  int set_allSettingsAndFiles( byte[] settings)
uwp   async Task<int> set_allSettingsAndFiles( byte[] settings)
py    set_allSettingsAndFiles( settings)
php   function set_allSettingsAndFiles( $settings)
es    async set_allSettingsAndFiles( settings)
dnp   int set_allSettingsAndFiles( byte[] settings)
cp    int set_allSettingsAndFiles( string settings)
cmd   YModule target set_allSettingsAndFiles settings
```

This method is useful to restore all the logical names and calibrations parameters, uploaded files etc. of a device from a backup. Remember to call the `saveToFlash()` method of the module if the modifications must be kept.

Parameters :

settings a binary buffer with all the settings.

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_beacon() YModule
module→setBeacon()

Turns on or off the module localization beacon.

```
js    function set_beacon( newval)
cpp   int set_beacon( Y_BEACON_enum newval)
m     -(int) setBeacon : (Y_BEACON_enum) newval
pas   integer set_beacon( newval: Integer): integer
vb    function set_beacon( ByVal newval As Integer) As Integer
cs    int set_beacon( int newval)
java  int set_beacon( int newval)
uwp   async Task<int> set_beacon( int newval)
py    set_beacon( newval)
php   function set_beacon( $newval)
es    async set_beacon( newval)
dnp   int set_beacon( int newval)
cp    int set_beacon( int newval)
cmd   YModule target set_beacon newval
```

Parameters :

newval either `Y_BEACON_OFF` or `Y_BEACON_ON`

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_logicalName() module→setLogicalName() YModule

Changes the logical name of the module.

js	function **set_logicalName**(**newval**)
cpp	int **set_logicalName**(string **newval**)
m	-(int) setLogicalName : (NSString*) **newval**
pas	integer **set_logicalName**(**newval**: string): integer
vb	function **set_logicalName**(ByVal **newval** As String) As Integer
cs	int **set_logicalName**(string **newval**)
java	int **set_logicalName**(String **newval**)
uwp	async Task<int> **set_logicalName**(string **newval**)
py	**set_logicalName**(**newval**)
php	function **set_logicalName**(**$newval**)
es	async **set_logicalName**(**newval**)
dnp	int **set_logicalName**(string **newval**)
cp	int **set_logicalName**(string **newval**)
cmd	YModule **target set_logicalName newval**

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the module

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_luminosity() YModule
module→setLuminosity()

Changes the luminosity of the module informative leds.

```
js    function set_luminosity( newval)
cpp   int set_luminosity( int newval)
m     -(int) setLuminosity : (int) newval
pas   integer set_luminosity( newval: LongInt): integer
vb    function set_luminosity( ByVal newval As Integer) As Integer
cs    int set_luminosity( int newval)
java  int set_luminosity( int newval)
uwp   async Task<int> set_luminosity( int newval)
py    set_luminosity( newval)
php   function set_luminosity( $newval)
es    async set_luminosity( newval)
dnp   int set_luminosity( int newval)
cp    int set_luminosity( int newval)
cmd   YModule target set_luminosity newval
```

The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer corresponding to the luminosity of the module informative leds

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_userData() YModule
module→setUserData()

Stores a user context provided as argument in the userData attribute of the function.

```
js    function set_userData( data)
cpp   void set_userData( void * data)
m     -(void) setUserData : (id) data
pas   set_userData( data: Tobject)
vb    procedure set_userData( ByVal data As Object)
cs    void set_userData( object data)
java  void set_userData( Object data)
py    set_userData( data)
php   function set_userData( $data)
es    async set_userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

module→set_userVar() YModule
module→setUserVar()

Stores a 32 bit value in the device RAM.

```
js    function set_userVar( newval)
cpp   int set_userVar( int newval)
m     -(int) setUserVar : (int) newval
pas   integer set_userVar( newval: LongInt): integer
vb    function set_userVar( ByVal newval As Integer) As Integer
cs    int set_userVar( int newval)
java  int set_userVar( int newval)
uwp   async Task<int> set_userVar( int newval)
py    set_userVar( newval)
php   function set_userVar( $newval)
es    async set_userVar( newval)
dnp   int set_userVar( int newval)
cp    int set_userVar( int newval)
cmd   YModule target set_userVar newval
```

This attribute is at programmer disposal, should he need to store a state variable. On startup and after a device reboot, the value is always reset to zero.

Parameters :

newval an integer

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→triggerConfigChangeCallback() YModule

Triggers a configuration change callback, to check if they are supported or not.

js	function **triggerConfigChangeCallback**( )
cpp	int **triggerConfigChangeCallback**( )
m	-(int) **triggerConfigChangeCallback**
pas	LongInt **triggerConfigChangeCallback**( ): LongInt
vb	function **triggerConfigChangeCallback**( ) As Integer
cs	int **triggerConfigChangeCallback**( )
java	int **triggerConfigChangeCallback**( )
uwp	async Task<int> **triggerConfigChangeCallback**( )
py	**triggerConfigChangeCallback**( )
php	function **triggerConfigChangeCallback**( )
es	async **triggerConfigChangeCallback**( )
dnp	int **triggerConfigChangeCallback**( )
cp	int **triggerConfigChangeCallback**( )
cmd	YModule **target triggerConfigChangeCallback**

module→triggerFirmwareUpdate() YModule

Schedules a module reboot into special firmware update mode.

```
js    function triggerFirmwareUpdate( secBeforeReboot)
cpp   int triggerFirmwareUpdate( int secBeforeReboot)
m     -(int) triggerFirmwareUpdate : (int) secBeforeReboot
pas   LongInt triggerFirmwareUpdate( secBeforeReboot: LongInt): LongInt
vb    function triggerFirmwareUpdate( ByVal secBeforeReboot As Integer) As Integer
cs    int triggerFirmwareUpdate( int secBeforeReboot)
java  int triggerFirmwareUpdate( int secBeforeReboot)
uwp   async Task<int> triggerFirmwareUpdate( int secBeforeReboot)
py    triggerFirmwareUpdate( secBeforeReboot)
php   function triggerFirmwareUpdate( $secBeforeReboot)
es    async triggerFirmwareUpdate( secBeforeReboot)
dnp   int triggerFirmwareUpdate( int secBeforeReboot)
cp    int triggerFirmwareUpdate( int secBeforeReboot)
cmd   YModule target triggerFirmwareUpdate secBeforeReboot
```

Parameters :

secBeforeReboot number of seconds before rebooting

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→updateFirmware() YModule

Prepares a firmware update of the module.

js	function **updateFirmware**(**path**)
cpp	YFirmwareUpdate **updateFirmware**(string **path**)
m	-(YFirmwareUpdate*) **updateFirmware** : (NSString*) **path**
pas	TYFirmwareUpdate **updateFirmware**(**path**: string): TYFirmwareUpdate
vb	function **updateFirmware**(ByVal **path** As String) As YFirmwareUpdate
cs	YFirmwareUpdate **updateFirmware**(string **path**)
java	YFirmwareUpdate **updateFirmware**(String **path**)
uwp	async Task<YFirmwareUpdate> **updateFirmware**(string **path**)
py	**updateFirmware**(**path**)
php	function **updateFirmware**(**$path**)
es	async **updateFirmware**(**path**)
dnp	YFirmwareUpdateProxy **updateFirmware**(string **path**)
cp	YFirmwareUpdateProxy* **updateFirmware**(string **path**)
cmd	YModule **target updateFirmware path**

This method returns a `YFirmwareUpdate` object which handles the firmware update process.

Parameters :

path the path of the `.byn` file to use.

Returns :

a `YFirmwareUpdate` object or NULL on error.

module→updateFirmwareEx() YModule

Prepares a firmware update of the module.

```
js    function updateFirmwareEx( path, force)
cpp   YFirmwareUpdate updateFirmwareEx( string path, bool force)
m     -(YFirmwareUpdate*) updateFirmwareEx : (NSString*) path
                                           : (bool) force
pas   TYFirmwareUpdate updateFirmwareEx( path: string, force: boolean): TYFirmwareUpdate
vb    function updateFirmwareEx( ByVal path As String,
                                 ByVal force As Boolean) As YFirmwareUpdate
cs    YFirmwareUpdate updateFirmwareEx( string path, bool force)
java  YFirmwareUpdate updateFirmwareEx( String path, boolean force)
uwp   async Task<YFirmwareUpdate> updateFirmwareEx( string path, bool force)
py    updateFirmwareEx( path, force)
php   function updateFirmwareEx( $path, $force)
es    async updateFirmwareEx( path, force)
dnp   YFirmwareUpdateProxy updateFirmwareEx( string path, bool force)
cp    YFirmwareUpdateProxy* updateFirmwareEx( string path,
                                              bool force)
cmd   YModule target updateFirmwareEx path force
```

This method returns a `YFirmwareUpdate` object which handles the firmware update process.

Parameters :

path the path of the `.byn` file to use.

force true to force the firmware update even if some prerequisites appear not to be met

Returns :

a `YFirmwareUpdate` object or NULL on error.

module→wait_async() YModule

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js  function wait_async( callback, context)
es  wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

24.3. Class YSerialPort

Serial port control interface, available for instance in the Yocto-RS232, the Yocto-RS485-V2 or the Yocto-Serial

The `YSerialPort` class allows you to fully drive a Yoctopuce serial port. It can be used to send and receive data, and to configure communication parameters (baud rate, bit count, parity, flow control and protocol). Note that Yoctopuce serial ports are not exposed as virtual COM ports. They are meant to be used in the same way as all Yoctopuce devices.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_serialport.js'></script>
cpp	#include "yocto_serialport.h"
m	#import "yocto_serialport.h"
pas	uses yocto_serialport;
vb	yocto_serialport.vb
cs	yocto_serialport.cs
java	import com.yoctopuce.YoctoAPI.YSerialPort;
uwp	import com.yoctopuce.YoctoAPI.YSerialPort;
py	from yocto_serialport import *
php	require_once('yocto_serialport.php');
es	in HTML: <script src="../../lib/yocto_serialport.js"></script> in node.js: require('yoctolib-es2017/yocto_serialport.js');
dnp	import YoctoProxyAPI.YSerialPortProxy
cp	#include "yocto_serialport_proxy.h"
vi	YSerialPort.vi
ml	import YoctoProxyAPI.YSerialPortProxy

Global functions
YSerialPort.FindSerialPort(**func**) Retrieves a serial port for a given identifier.
YSerialPort.FindSerialPortInContext(**yctx**, **func**) Retrieves a serial port for a given identifier in a YAPI context.
YSerialPort.FirstSerialPort() Starts the enumeration of serial ports currently accessible.
YSerialPort.FirstSerialPortInContext(**yctx**) Starts the enumeration of serial ports currently accessible.
YSerialPort.GetSimilarFunctions() Enumerates all functions of type SerialPort available on the devices currently reachable by the library, and returns their unique hardware ID.
`YSerialPort` properties
serialport→AdvertisedValue *[read-only]* Short string representing the current state of the function.
serialport→FriendlyName *[read-only]* Global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.
serialport→FunctionId *[read-only]* Hardware identifier of the serial port, without reference to the module.
serialport→HardwareId *[read-only]*

Unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`.

serialport→IsOnline *[read-only]*

Checks if the function is currently reachable.

serialport→JobMaxSize *[read-only]*

Maximum size allowed for job files.

serialport→JobMaxTask *[read-only]*

Maximum number of tasks in a job that the device can handle.

serialport→LogicalName *[writable]*

Logical name of the function.

serialport→Protocol *[writable]*

Type of protocol used over the serial line, as a string.

serialport→SerialMode *[writable]*

Serial port communication parameters, as a string such as "9600,8N1".

serialport→SerialNumber *[read-only]*

Serial number of the module, as set by the factory.

serialport→StartupJob *[writable]*

Job file to use when the device is powered on.

serialport→VoltageLevel *[writable]*

Voltage level used on the serial line.

`YSerialPort` methods

serialport→clearCache()

Invalidates the cache.

serialport→describe()

Returns a short text that describes unambiguously the instance of the serial port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

serialport→get_CTS()

Reads the level of the CTS line.

serialport→get_advertisedValue()

Returns the current value of the serial port (no more than 6 characters).

serialport→get_currentJob()

Returns the name of the job file currently in use.

serialport→get_errCount()

Returns the total number of communication errors detected since last reset.

serialport→get_errorMessage()

Returns the error message of the latest error with the serial port.

serialport→get_errorType()

Returns the numerical error code of the latest error with the serial port.

serialport→get_friendlyName()

Returns a global identifier of the serial port in the format `MODULE_NAME.FUNCTION_NAME`.

serialport→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

serialport→get_functionId()

Returns the hardware identifier of the serial port, without reference to the module.

serialport→get_hardwareId()

Returns the unique hardware identifier of the serial port in the form `SERIAL.FUNCTIONID`.

serialport→get_jobMaxSize()
Returns maximum size allowed for job files.

serialport→get_jobMaxTask()
Returns the maximum number of tasks in a job that the device can handle.

serialport→get_lastMsg()
Returns the latest message fully received (for Line, Frame and Modbus protocols).

serialport→get_logicalName()
Returns the logical name of the serial port.

serialport→get_module()
Gets the `YModule` object for the device on which the function is located.

serialport→get_module_async(callback, context)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

serialport→get_protocol()
Returns the type of protocol used over the serial line, as a string.

serialport→get_rxCount()
Returns the total number of bytes received since last reset.

serialport→get_rxMsgCount()
Returns the total number of messages received since last reset.

serialport→get_serialMode()
Returns the serial port communication parameters, as a string such as "9600,8N1".

serialport→get_serialNumber()
Returns the serial number of the module, as set by the factory.

serialport→get_startupJob()
Returns the job file to use when the device is powered on.

serialport→get_txCount()
Returns the total number of bytes transmitted since last reset.

serialport→get_txMsgCount()
Returns the total number of messages send since last reset.

serialport→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

serialport→get_voltageLevel()
Returns the voltage level used on the serial line.

serialport→isOnline()
Checks if the serial port is currently reachable, without raising any error.

serialport→isOnline_async(callback, context)
Checks if the serial port is currently reachable, without raising any error (asynchronous version).

serialport→isReadOnly()
Test if the function is readOnly.

serialport→load(msValidity)
Preloads the serial port cache with a specified validity duration.

serialport→loadAttribute(attrName)
Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

serialport→load_async(msValidity, callback, context)
Preloads the serial port cache with a specified validity duration (asynchronous version).

serialport→modbusReadBits(slaveNo, pduAddr, nBits)

Reads one or more contiguous internal bits (or coil status) from a MODBUS serial device.

serialport→modbusReadInputBits(slaveNo, pduAddr, nBits)

Reads one or more contiguous input bits (or discrete inputs) from a MODBUS serial device.

serialport→modbusReadInputRegisters(slaveNo, pduAddr, nWords)

Reads one or more contiguous input registers (read-only registers) from a MODBUS serial device.

serialport→modbusReadRegisters(slaveNo, pduAddr, nWords)

Reads one or more contiguous internal registers (holding registers) from a MODBUS serial device.

serialport→modbusWriteAndReadRegisters(slaveNo, pduWriteAddr, values, pduReadAddr, nReadWords)

Sets several contiguous internal registers (holding registers) on a MODBUS serial device, then performs a contiguous read of a set of (possibly different) internal registers.

serialport→modbusWriteBit(slaveNo, pduAddr, value)

Sets a single internal bit (or coil) on a MODBUS serial device.

serialport→modbusWriteBits(slaveNo, pduAddr, bits)

Sets several contiguous internal bits (or coils) on a MODBUS serial device.

serialport→modbusWriteRegister(slaveNo, pduAddr, value)

Sets a single internal register (or holding register) on a MODBUS serial device.

serialport→modbusWriteRegisters(slaveNo, pduAddr, values)

Sets several contiguous internal registers (or holding registers) on a MODBUS serial device.

serialport→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

serialport→nextSerialPort()

Continues the enumeration of serial ports started using `yFirstSerialPort()`.

serialport→queryHex(hexString, maxWait)

Sends a binary message to the serial port, and reads the reply, if any.

serialport→queryLine(query, maxWait)

Sends a text line query to the serial port, and reads the reply, if any.

serialport→queryMODBUS(slaveNo, pduBytes)

Sends a message to a specified MODBUS slave connected to the serial port, and reads the reply, if any.

serialport→readArray(nChars)

Reads data from the receive buffer as a list of bytes, starting at current stream position.

serialport→readBin(nChars)

Reads data from the receive buffer as a binary buffer, starting at current stream position.

serialport→readByte()

Reads one byte from the receive buffer, starting at current stream position.

serialport→readHex(nBytes)

Reads data from the receive buffer as a hexadecimal string, starting at current stream position.

serialport→readLine()

Reads a single line (or message) from the receive buffer, starting at current stream position.

serialport→readMessages(pattern, maxWait)

Searches for incoming messages in the serial port receive buffer matching a given pattern, starting at current position.

serialport→readStr(nChars)

Reads data from the receive buffer as a string, starting at current stream position.

serialport→read_avail()

Returns the number of bytes available to read in the input buffer starting from the current absolute stream position pointer of the API object.

serialport→read_seek(absPos)

Changes the current internal stream position to the specified value.

serialport→read_tell()

Returns the current absolute stream position pointer of the API object.

serialport→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

serialport→reset()

Clears the serial port buffer and resets counters to zero.

serialport→selectJob(jobfile)

Load and start processing the specified job file.

serialport→set_RTS(val)

Manually sets the state of the RTS line.

serialport→set_currentJob(newval)

Selects a job file to run immediately.

serialport→set_logicalName(newval)

Changes the logical name of the serial port.

serialport→set_protocol(newval)

Changes the type of protocol used over the serial line.

serialport→set_serialMode(newval)

Changes the serial port communication parameters, with a string such as "9600,8N1".

serialport→set_startupJob(newval)

Changes the job to use when the device is powered on.

serialport→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

serialport→set_voltageLevel(newval)

Changes the voltage type used on the serial line.

serialport→snoopMessages(maxWait)

Retrieves messages (both direction) in the serial port buffer, starting at current position.

serialport→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

serialport→uploadJob(jobfile, jsonDef)

Saves the job definition string (JSON data) into a job file.

serialport→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

serialport→writeArray(byteList)

Sends a byte sequence (provided as a list of bytes) to the serial port.

serialport→writeBin(buff)

Sends a binary buffer to the serial port, as is.

serialport→writeByte(code)

Sends a single byte to the serial port.

serialport→writeHex(hexString)

Sends a byte sequence (provided as a hexadecimal string) to the serial port.

serialport→writeLine(text)

Sends an ASCII string to the serial port, followed by a line break (CR LF).

serialport→writeMODBUS(hexString)

Sends a MODBUS message (provided as a hexadecimal string) to the serial port.

serialport→writeStr(text)

Sends an ASCII string to the serial port, as is.

YSerialPort.FindSerialPort() YSerialPort
YSerialPort.FindSerialPort()

Retrieves a serial port for a given identifier.

```
js    function yFindSerialPort( func)
cpp   YSerialPort* yFindSerialPort( string func)
m     +(YSerialPort*) FindSerialPort : (NSString*) func
pas   TYSerialPort yFindSerialPort( func: string): TYSerialPort
vb    function yFindSerialPort( ByVal func As String) As YSerialPort
cs    static YSerialPort FindSerialPort( string func)
java  static YSerialPort FindSerialPort( String func)
uwp   static YSerialPort FindSerialPort( string func)
py    FindSerialPort( func)
php   function yFindSerialPort( $func)
es    static FindSerialPort( func)
dnp   static YSerialPortProxy FindSerialPort( string func)
cp    static YSerialPortProxy * FindSerialPort( string 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 serial port is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YSerialPort.isOnline()` to test if the serial port is indeed online at a given time. In case of ambiguity when looking for a serial port by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the serial port, for instance `RS232MK1.serialPort`.

Returns :

a `YSerialPort` object allowing you to drive the serial port.

YSerialPort.FindSerialPortInContext() YSerialPort.FindSerialPortInContext() — YSerialPort

Retrieves a serial port for a given identifier in a YAPI context.

```
java  static YSerialPort FindSerialPortInContext( YAPIContext yctx,
                                                  String func)
uwp   static YSerialPort FindSerialPortInContext( YAPIContext yctx,
                                                  string func)
es    static FindSerialPortInContext( yctx, func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the serial port is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YSerialPort.isOnline()` to test if the serial port is indeed online at a given time. In case of ambiguity when looking for a serial port by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the serial port, for instance `RS232MK1.serialPort`.

Returns :

a `YSerialPort` object allowing you to drive the serial port.

YSerialPort.FirstSerialPort() YSerialPort

YSerialPort.FirstSerialPort()

Starts the enumeration of serial ports currently accessible.

js	function **yFirstSerialPort**( )
cpp	YSerialPort * **yFirstSerialPort**( )
m	+(YSerialPort*) **FirstSerialPort**
pas	TYSerialPort **yFirstSerialPort**( ): TYSerialPort
vb	function **yFirstSerialPort**( ) As YSerialPort
cs	static YSerialPort **FirstSerialPort**( )
java	static YSerialPort **FirstSerialPort**( )
uwp	static YSerialPort **FirstSerialPort**( )
py	**FirstSerialPort**( )
php	function **yFirstSerialPort**( )
es	static **FirstSerialPort**( )

Use the method `YSerialPort.nextSerialPort()` to iterate on next serial ports.

Returns :

a pointer to a `YSerialPort` object, corresponding to the first serial port currently online, or a `null` pointer if there are none.

YSerialPort.FirstSerialPortInContext() YSerialPort

YSerialPort.FirstSerialPortInContext()

Starts the enumeration of serial ports currently accessible.

java	static YSerialPort **FirstSerialPortInContext**(YAPIContext **yctx**)
uwp	static YSerialPort **FirstSerialPortInContext**(YAPIContext **yctx**)
es	static **FirstSerialPortInContext**(**yctx**)

Use the method `YSerialPort.nextSerialPort()` to iterate on next serial ports.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YSerialPort` object, corresponding to the first serial port currently online, or a `null` pointer if there are none.

YSerialPort.GetSimilarFunctions() YSerialPort

YSerialPort.GetSimilarFunctions()

Enumerates all functions of type SerialPort available on the devices currently reachable by the library, and returns their unique hardware ID.

dnp static new string[] **GetSimilarFunctions**( )

cp static vector<string> **GetSimilarFunctions**( )

Each of these IDs can be provided as argument to the method `YSerialPort.FindSerialPort` to obtain an object that can control the corresponding device.

Returns :

an array of strings, each string containing the unique hardwareId of a device function currently connected.

serialport→AdvertisedValue YSerialPort

Short string representing the current state of the function.

dnp string **AdvertisedValue**

serialport→FriendlyName YSerialPort

Global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

dnp string **FriendlyName**

The returned string uses the logical names of the module and of the function if they are defined, otherwise the serial number of the module and the hardware identifier of the function (for example: `MyCustomName.relay1`)

serialport→FunctionId YSerialPort

Hardware identifier of the serial port, without reference to the module.

dnp string **FunctionId**

For example `relay1`

serialport→HardwareId YSerialPort

Unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`.

dnp string **HardwareId**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example `RELAYLO1-123456.relay1`).

serialport→IsOnline YSerialPort

Checks if the function is currently reachable.

dnp bool **IsOnline**

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the function.

serialport→JobMaxSize YSerialPort

Maximum size allowed for job files.

dnp int **JobMaxSize**

serialport→JobMaxTask YSerialPort

Maximum number of tasks in a job that the device can handle.

dnp int **JobMaxTask**

serialport→LogicalName YSerialPort

Logical name of the function.

dnp string **LogicalName**

Writable. You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

serialport→Protocol YSerialPort

Type of protocol used over the serial line, as a string.

dnp string **Protocol**

Possible values are "Line" for ASCII messages separated by CR and/or LF, "Frame:[timeout]ms" for binary messages separated by a delay time, "Modbus-ASCII" for MODBUS messages in ASCII mode, "Modbus-RTU" for MODBUS messages in RTU mode, "Wiegand-ASCII" for Wiegand messages in ASCII mode, "Wiegand-26","Wiegand-34", etc for Wiegand messages in byte mode, "Char" for a continuous ASCII stream or "Byte" for a continuous binary stream.

Writable. Changes the type of protocol used over the serial line. Possible values are "Line" for ASCII messages separated by CR and/or LF, "Frame:[timeout]ms" for binary messages separated by a delay time, "Modbus-ASCII" for MODBUS messages in ASCII mode, "Modbus-RTU" for MODBUS messages in RTU mode, "Wiegand-ASCII" for Wiegand messages in ASCII mode, "Wiegand-26","Wiegand-34", etc for Wiegand messages in byte mode, "Char" for a continuous ASCII stream or "Byte" for a continuous binary stream. The suffix "/[wait]ms" can be added to reduce the transmit rate so that there is always at lest the specified number of milliseconds between each bytes sent. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

serialport→SerialMode YSerialPort

Serial port communication parameters, as a string such as "9600,8N1".

dnp string **SerialMode**

The string includes the baud rate, the number of data bits, the parity, and the number of stop bits. An optional suffix is included if flow control is active: "CtsRts" for hardware handshake, "XOnXOff" for logical flow control and "Simplex" for acquiring a shared bus using the RTS line (as used by some RS485 adapters for instance).

Writable. Changes the serial port communication parameters, with a string such as "9600,8N1". The string includes the baud rate, the number of data bits, the parity, and the number of stop bits. An optional suffix can be added to enable flow control: "CtsRts" for hardware handshake, "XOnXOff" for logical flow control and "Simplex" for acquiring a shared bus using the RTS line (as used by some RS485 adapters for instance). Remember to call the `saveToFlash()` method of the module if the modification must be kept.

serialport→SerialNumber YSerialPort

Serial number of the module, as set by the factory.

dnp string **SerialNumber**

serialport→StartupJob YSerialPort

Job file to use when the device is powered on.

dnp string **StartupJob**

Writable. Changes the job to use when the device is powered on. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

serialport→VoltageLevel YSerialPort

Voltage level used on the serial line.

dnp int **VoltageLevel**

Possible values:

`Y_VOLTAGELEVEL_INVALID`	= 0
`Y_VOLTAGELEVEL_OFF`	= 1
`Y_VOLTAGELEVEL_TTL3V`	= 2
`Y_VOLTAGELEVEL_TTL3VR`	= 3
`Y_VOLTAGELEVEL_TTL5V`	= 4
`Y_VOLTAGELEVEL_TTL5VR`	= 5
`Y_VOLTAGELEVEL_RS232`	= 6
`Y_VOLTAGELEVEL_RS485`	= 7
`Y_VOLTAGELEVEL_TTL1V8`	= 8

Writable. Changes the voltage type used on the serial line. Valid values will depend on the Yoctopuce device model featuring the serial port feature. Check your device documentation to find out which values are valid for that specific model. Trying to set an invalid value will have no effect. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

serialport→clearCache() YSerialPort

Invalidates the cache.

js	function **clearCache**( )
cpp	void **clearCache**( )
m	-(void) **clearCache**
pas	**clearCache**( )
vb	procedure **clearCache**( )
cs	void **clearCache**( )
java	void **clearCache**( )
py	**clearCache**( )
php	function **clearCache**( )
es	async **clearCache**( )

Invalidates the cache of the serial port attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

serialport→describe() YSerialPort

Returns a short text that describes unambiguously the instance of the serial port in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
js    function describe( )
cpp   string describe( )
m     -(NSString*) describe
pas   string describe( ): string
vb    function describe( ) As String
cs    string describe( )
java  String describe( )
py    describe( )
php   function describe( )
es    async describe( )
```

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :
a string that describes the serial port (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

serialport→get_CTS() YSerialPort
serialport→CTS()

Reads the level of the CTS line.

```
js    function get_CTS( )
cpp   int get_CTS( )
m     -(int) CTS
pas   LongInt get_CTS( ): LongInt
vb    function get_CTS( ) As Integer
cs    int get_CTS( )
java  int get_CTS( )
uwp   async Task<int> get_CTS( )
py    get_CTS( )
php   function get_CTS( )
es    async get_CTS( )
dnp   int get_CTS( )
cp    int get_CTS( )
cmd   YSerialPort target get_CTS
```

The CTS line is usually driven by the RTS signal of the connected serial device.

Returns :
1 if the CTS line is high, 0 if the CTS line is low.

On failure, throws an exception or returns a negative error code.

serialport→get_advertisedValue() YSerialPort
serialport→advertisedValue()

Returns the current value of the serial port (no more than 6 characters).

```
js    function get_advertisedValue( )
cpp   string get_advertisedValue( )
m     -(NSString*) advertisedValue
pas   string get_advertisedValue( ): string
vb    function get_advertisedValue( ) As String
cs    string get_advertisedValue( )
java  String get_advertisedValue( )
uwp   async Task<string> get_advertisedValue( )
py    get_advertisedValue( )
php   function get_advertisedValue( )
es    async get_advertisedValue( )
dnp   string get_advertisedValue( )
cp    string get_advertisedValue( )
cmd   YSerialPort target get_advertisedValue
```

Returns :

a string corresponding to the current value of the serial port (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

serialport→get_currentJob() YSerialPort
serialport→currentJob()

Returns the name of the job file currently in use.

```
js    function get_currentJob( )
cpp   string get_currentJob( )
m     -(NSString*) currentJob
pas   string get_currentJob( ): string
vb    function get_currentJob( ) As String
cs    string get_currentJob( )
java  String get_currentJob( )
uwp   async Task<string> get_currentJob( )
py    get_currentJob( )
php   function get_currentJob( )
es    async get_currentJob( )
dnp   string get_currentJob( )
cp    string get_currentJob( )
cmd   YSerialPort target get_currentJob
```

Returns :
a string corresponding to the name of the job file currently in use

On failure, throws an exception or returns `Y_CURRENTJOB_INVALID`.

serialport→get_errCount() YSerialPort
serialport→errCount()

Returns the total number of communication errors detected since last reset.

js	function **get_errCount**( )
cpp	int **get_errCount**( )
m	-(int) errCount
pas	LongInt **get_errCount**( ): LongInt
vb	function **get_errCount**( ) As Integer
cs	int **get_errCount**( )
java	int **get_errCount**( )
uwp	async Task<int> **get_errCount**( )
py	**get_errCount**( )
php	function **get_errCount**( )
es	async **get_errCount**( )
dnp	int **get_errCount**( )
cp	int **get_errCount**( )
cmd	YSerialPort **target get_errCount**

Returns :
an integer corresponding to the total number of communication errors detected since last reset

On failure, throws an exception or returns `Y_ERRCOUNT_INVALID`.

serialport→get_errorMessage() YSerialPort
serialport→errorMessage()

Returns the error message of the latest error with the serial port.

```
js    function get_errorMessage( )
cpp   string get_errorMessage( )
m     -(NSString*) errorMessage
pas   string get_errorMessage( ): string
vb    function get_errorMessage( ) As String
cs    string get_errorMessage( )
java  String get_errorMessage( )
py    get_errorMessage( )
php   function get_errorMessage( )
es    get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the serial port object

serialport→get_errorType() YSerialPort
serialport→errorType()

Returns the numerical error code of the latest error with the serial port.

js	function **get_errorType**( )
cpp	YRETCODE **get_errorType**( )
m	-(YRETCODE) errorType
pas	YRETCODE **get_errorType**( ): YRETCODE
vb	function **get_errorType**( ) As YRETCODE
cs	YRETCODE **get_errorType**( )
java	int **get_errorType**( )
py	**get_errorType**( )
php	function **get_errorType**( )
es	**get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the serial port object

serialport→get_friendlyName() YSerialPort
serialport→friendlyName()

Returns a global identifier of the serial port in the format `MODULE_NAME.FUNCTION_NAME`.

js	function **get_friendlyName**( )
cpp	string **get_friendlyName**( )
m	-(NSString*) friendlyName
cs	string **get_friendlyName**( )
java	String **get_friendlyName**( )
py	**get_friendlyName**( )
php	function **get_friendlyName**( )
es	async **get_friendlyName**( )
dnp	string **get_friendlyName**( )
cp	string **get_friendlyName**( )

The returned string uses the logical names of the module and of the serial port if they are defined, otherwise the serial number of the module and the hardware identifier of the serial port (for example: `MyCustomName.relay1`)

Returns :
a string that uniquely identifies the serial port using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

serialport→get_functionDescriptor() YSerialPort
serialport→functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

js	function **get_functionDescriptor**( )
cpp	YFUN_DESCR **get_functionDescriptor**( )
m	-(YFUN_DESCR) functionDescriptor
pas	YFUN_DESCR **get_functionDescriptor**( ): YFUN_DESCR
vb	function **get_functionDescriptor**( ) As YFUN_DESCR
cs	YFUN_DESCR **get_functionDescriptor**( )
java	String **get_functionDescriptor**( )
py	**get_functionDescriptor**( )
php	function **get_functionDescriptor**( )
es	async **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :
an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

serialport→get_functionId() YSerialPort
serialport→functionId()

Returns the hardware identifier of the serial port, without reference to the module.

js	function **get_functionId**( )
cpp	string **get_functionId**( )
m	-(NSString*) functionId
vb	function **get_functionId**( ) As String
cs	string **get_functionId**( )
java	String **get_functionId**( )
py	**get_functionId**( )
php	function **get_functionId**( )
es	async **get_functionId**( )
dnp	string **get_functionId**( )
cp	string **get_functionId**( )

For example `relay1`

Returns :
a string that identifies the serial port (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

serialport→get_hardwareId() YSerialPort
serialport→hardwareId()

Returns the unique hardware identifier of the serial port in the form `SERIAL.FUNCTIONID`.

js	function **get_hardwareId**( )
cpp	string **get_hardwareId**( )
m	-(NSString*) hardwareId
vb	function **get_hardwareId**( ) As String
cs	string **get_hardwareId**( )
java	String **get_hardwareId**( )
py	**get_hardwareId**( )
php	function **get_hardwareId**( )
es	async **get_hardwareId**( )
dnp	string **get_hardwareId**( )
cp	string **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the serial port (for example `RELAYLO1-123456.relay1`).

Returns :
a string that uniquely identifies the serial port (ex: `RELAYLO1-123456.relay1`)

On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

serialport→get_jobMaxSize() YSerialPort
serialport→jobMaxSize()

Returns maximum size allowed for job files.

js	function **get_jobMaxSize**( )
cpp	int **get_jobMaxSize**( )
m	-(int) jobMaxSize
pas	LongInt **get_jobMaxSize**( ): LongInt
vb	function **get_jobMaxSize**( ) As Integer
cs	int **get_jobMaxSize**( )
java	int **get_jobMaxSize**( )
uwp	async Task<int> **get_jobMaxSize**( )
py	**get_jobMaxSize**( )
php	function **get_jobMaxSize**( )
es	async **get_jobMaxSize**( )
dnp	int **get_jobMaxSize**( )
cp	int **get_jobMaxSize**( )
cmd	YSerialPort **target get_jobMaxSize**

Returns :
an integer corresponding to maximum size allowed for job files

On failure, throws an exception or returns `Y_JOBMAXSIZE_INVALID`.

serialport→get_jobMaxTask() YSerialPort
serialport→jobMaxTask()

Returns the maximum number of tasks in a job that the device can handle.

js	function **get_jobMaxTask**( )
cpp	int **get_jobMaxTask**( )
m	-(int) jobMaxTask
pas	LongInt **get_jobMaxTask**( ): LongInt
vb	function **get_jobMaxTask**( ) As Integer
cs	int **get_jobMaxTask**( )
java	int **get_jobMaxTask**( )
uwp	async Task<int> **get_jobMaxTask**( )
py	**get_jobMaxTask**( )
php	function **get_jobMaxTask**( )
es	async **get_jobMaxTask**( )
dnp	int **get_jobMaxTask**( )
cp	int **get_jobMaxTask**( )
cmd	YSerialPort **target get_jobMaxTask**

Returns :
an integer corresponding to the maximum number of tasks in a job that the device can handle

On failure, throws an exception or returns `Y_JOBMAXTASK_INVALID`.

serialport→get_lastMsg() YSerialPort
serialport→lastMsg()

Returns the latest message fully received (for Line, Frame and Modbus protocols).

```
js    function get_lastMsg( )
cpp   string get_lastMsg( )
m     -(NSString*) lastMsg
pas   string get_lastMsg( ): string
vb    function get_lastMsg( ) As String
cs    string get_lastMsg( )
java  String get_lastMsg( )
uwp   async Task<string> get_lastMsg( )
py    get_lastMsg( )
php   function get_lastMsg( )
es    async get_lastMsg( )
dnp   string get_lastMsg( )
cp    string get_lastMsg( )
cmd   YSerialPort target get_lastMsg
```

Returns :
a string corresponding to the latest message fully received (for Line, Frame and Modbus protocols)

On failure, throws an exception or returns `Y_LASTMSG_INVALID`.

serialport→get_logicalName() YSerialPort
serialport→logicalName()

Returns the logical name of the serial port.

js	function **get_logicalName**( )
cpp	string **get_logicalName**( )
m	-(NSString*) logicalName
pas	string **get_logicalName**( ): string
vb	function **get_logicalName**( ) As String
cs	string **get_logicalName**( )
java	String **get_logicalName**( )
uwp	async Task<string> **get_logicalName**( )
py	**get_logicalName**( )
php	function **get_logicalName**( )
es	async **get_logicalName**( )
dnp	string **get_logicalName**( )
cp	string **get_logicalName**( )
cmd	YSerialPort **target get_logicalName**

Returns :
a string corresponding to the logical name of the serial port.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

serialport→get_module() YSerialPort
serialport→module()

Gets the `YModule` object for the device on which the function is located.

js	function **get_module**( )
cpp	YModule * **get_module**( )
m	-(YModule*) module
pas	TYModule **get_module**( ): TYModule
vb	function **get_module**( ) As YModule
cs	YModule **get_module**( )
java	YModule **get_module**( )
py	**get_module**( )
php	function **get_module**( )
es	async **get_module**( )
dnp	YModuleProxy **get_module**( )
cp	YModuleProxy * **get_module**( )

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

Returns :
an instance of `YModule`

serialport→get_module_async() YSerialPort
serialport→module_async()

Gets the `YModule` object for the device on which the function is located (asynchronous version).

```
js function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as online.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking Firefox JavaScript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous JavasSript calls for more details.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

serialport→get_protocol() YSerialPort
serialport→protocol()

Returns the type of protocol used over the serial line, as a string.

```
js    function get_protocol( )
cpp   string get_protocol( )
m     -(NSString*) protocol
pas   string get_protocol( ): string
vb    function get_protocol( ) As String
cs    string get_protocol( )
java  String get_protocol( )
uwp   async Task<string> get_protocol( )
py    get_protocol( )
php   function get_protocol( )
es    async get_protocol( )
dnp   string get_protocol( )
cp    string get_protocol( )
cmd   YSerialPort target get_protocol
```

Possible values are "Line" for ASCII messages separated by CR and/or LF, "Frame:[timeout]ms" for binary messages separated by a delay time, "Modbus-ASCII" for MODBUS messages in ASCII mode, "Modbus-RTU" for MODBUS messages in RTU mode, "Wiegand-ASCII" for Wiegand messages in ASCII mode, "Wiegand-26","Wiegand-34", etc for Wiegand messages in byte mode, "Char" for a continuous ASCII stream or "Byte" for a continuous binary stream.

Returns :
a string corresponding to the type of protocol used over the serial line, as a string

On failure, throws an exception or returns `Y_PROTOCOL_INVALID`.

serialport→get_rxCount() YSerialPort
serialport→rxCount()

Returns the total number of bytes received since last reset.

js	function **get_rxCount**( )
cpp	int **get_rxCount**( )
m	-(int) rxCount
pas	LongInt **get_rxCount**( ): LongInt
vb	function **get_rxCount**( ) As Integer
cs	int **get_rxCount**( )
java	int **get_rxCount**( )
uwp	async Task<int> **get_rxCount**( )
py	**get_rxCount**( )
php	function **get_rxCount**( )
es	async **get_rxCount**( )
dnp	int **get_rxCount**( )
cp	int **get_rxCount**( )
cmd	YSerialPort **target get_rxCount**

Returns :
an integer corresponding to the total number of bytes received since last reset

On failure, throws an exception or returns `Y_RXCOUNT_INVALID`.

serialport→get_rxMsgCount() YSerialPort
serialport→rxMsgCount()

Returns the total number of messages received since last reset.

```
js    function get_rxMsgCount( )
cpp   int get_rxMsgCount( )
m     -(int) rxMsgCount
pas   LongInt get_rxMsgCount( ): LongInt
vb    function get_rxMsgCount( ) As Integer
cs    int get_rxMsgCount( )
java  int get_rxMsgCount( )
uwp   async Task<int> get_rxMsgCount( )
py    get_rxMsgCount( )
php   function get_rxMsgCount( )
es    async get_rxMsgCount( )
dnp   int get_rxMsgCount( )
cp    int get_rxMsgCount( )
cmd   YSerialPort target get_rxMsgCount
```

Returns :
an integer corresponding to the total number of messages received since last reset

On failure, throws an exception or returns `Y_RXMSGCOUNT_INVALID`.

serialport→get_serialMode() YSerialPort
serialport→serialMode()

Returns the serial port communication parameters, as a string such as "9600,8N1".

js	function **get_serialMode**( )
cpp	string **get_serialMode**( )
m	-(NSString*) serialMode
pas	string **get_serialMode**( ): string
vb	function **get_serialMode**( ) As String
cs	string **get_serialMode**( )
java	String **get_serialMode**( )
uwp	async Task<string> **get_serialMode**( )
py	**get_serialMode**( )
php	function **get_serialMode**( )
es	async **get_serialMode**( )
dnp	string **get_serialMode**( )
cp	string **get_serialMode**( )
cmd	YSerialPort **target get_serialMode**

The string includes the baud rate, the number of data bits, the parity, and the number of stop bits. An optional suffix is included if flow control is active: "CtsRts" for hardware handshake, "XOnXOff" for logical flow control and "Simplex" for acquiring a shared bus using the RTS line (as used by some RS485 adapters for instance).

Returns :
a string corresponding to the serial port communication parameters, as a string such as "9600,8N1"

On failure, throws an exception or returns `Y_SERIALMODE_INVALID`.

serialport→get_serialNumber() YSerialPort
serialport→serialNumber()

Returns the serial number of the module, as set by the factory.

js	function **get_serialNumber**( )
cpp	string **get_serialNumber**( )
m	-(NSString*) serialNumber
pas	string **get_serialNumber**( ): string
vb	function **get_serialNumber**( ) As String
cs	string **get_serialNumber**( )
java	String **get_serialNumber**( )
uwp	async Task<string> **get_serialNumber**( )
py	**get_serialNumber**( )
php	function **get_serialNumber**( )
es	async **get_serialNumber**( )
dnp	string **get_serialNumber**( )
cp	string **get_serialNumber**( )
cmd	YSerialPort **target get_serialNumber**

Returns :
a string corresponding to the serial number of the module, as set by the factory.

On failure, throws an exception or returns YModule.SERIALNUMBER_INVALID.

serialport→get_startupJob() YSerialPort
serialport→startupJob()

Returns the job file to use when the device is powered on.

js	function **get_startupJob**( )
cpp	string **get_startupJob**( )
m	-(NSString*) startupJob
pas	string **get_startupJob**( ): string
vb	function **get_startupJob**( ) As String
cs	string **get_startupJob**( )
java	String **get_startupJob**( )
uwp	async Task<string> **get_startupJob**( )
py	**get_startupJob**( )
php	function **get_startupJob**( )
es	async **get_startupJob**( )
dnp	string **get_startupJob**( )
cp	string **get_startupJob**( )
cmd	YSerialPort **target get_startupJob**

Returns :
a string corresponding to the job file to use when the device is powered on

On failure, throws an exception or returns `Y_STARTUPJOB_INVALID`.

serialport→get_txCount() YSerialPort
serialport→txCount()

Returns the total number of bytes transmitted since last reset.

```
js    function get_txCount( )
cpp   int get_txCount( )
m     -(int) txCount
pas   LongInt get_txCount( ): LongInt
vb    function get_txCount( ) As Integer
cs    int get_txCount( )
java  int get_txCount( )
uwp   async Task<int> get_txCount( )
py    get_txCount( )
php   function get_txCount( )
es    async get_txCount( )
dnp   int get_txCount( )
cp    int get_txCount( )
cmd   YSerialPort target get_txCount
```

Returns :
an integer corresponding to the total number of bytes transmitted since last reset

On failure, throws an exception or returns `Y_TXCOUNT_INVALID`.

serialport→get_txMsgCount() YSerialPort
serialport→txMsgCount()

Returns the total number of messages send since last reset.

js	function **get_txMsgCount**( )
cpp	int **get_txMsgCount**( )
m	-(int) txMsgCount
pas	LongInt **get_txMsgCount**( ): LongInt
vb	function **get_txMsgCount**( ) As Integer
cs	int **get_txMsgCount**( )
java	int **get_txMsgCount**( )
uwp	async Task<int> **get_txMsgCount**( )
py	**get_txMsgCount**( )
php	function **get_txMsgCount**( )
es	async **get_txMsgCount**( )
dnp	int **get_txMsgCount**( )
cp	int **get_txMsgCount**( )
cmd	YSerialPort **target get_txMsgCount**

Returns :
an integer corresponding to the total number of messages send since last reset

On failure, throws an exception or returns `Y_TXMSGCOUNT_INVALID`.

serialport→get_userData() YSerialPort
serialport→userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
js      function get_userData( )
cpp     void * get_userData( )
m       -(id) userData
pas     Tobject get_userData( ): Tobject
vb      function get_userData( ) As Object
cs      object get_userData( )
java    Object get_userData( )
py      get_userData( )
php     function get_userData( )
es      async get_userData( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

serialport→get_voltageLevel() YSerialPort
serialport→voltageLevel()

Returns the voltage level used on the serial line.

```
js    function get_voltageLevel( )
cpp   Y_VOLTAGELEVEL_enum get_voltageLevel( )
m     -(Y_VOLTAGELEVEL_enum) voltageLevel
pas   Integer get_voltageLevel( ): Integer
vb    function get_voltageLevel( ) As Integer
cs    int get_voltageLevel( )
java  int get_voltageLevel( )
uwp   async Task<int> get_voltageLevel( )
py    get_voltageLevel( )
php   function get_voltageLevel( )
es    async get_voltageLevel( )
dnp   int get_voltageLevel( )
cp    int get_voltageLevel( )
cmd   YSerialPort target get_voltageLevel
```

Returns :

a value among `Y_VOLTAGELEVEL_OFF`, `Y_VOLTAGELEVEL_TTL3V`, `Y_VOLTAGELEVEL_TTL3VR`, `Y_VOLTAGELEVEL_TTL5V`, `Y_VOLTAGELEVEL_TTL5VR`, `Y_VOLTAGELEVEL_RS232`, `Y_VOLTAGELEVEL_RS485` and `Y_VOLTAGELEVEL_TTL1V8` corresponding to the voltage level used on the serial line

On failure, throws an exception or returns `Y_VOLTAGELEVEL_INVALID`.

serialport→isOnline() YSerialPort

Checks if the serial port is currently reachable, without raising any error.

js	function **isOnline**( )
cpp	bool **isOnline**( )
m	-(BOOL) **isOnline**
pas	boolean **isOnline**( ): boolean
vb	function **isOnline**( ) As Boolean
cs	bool **isOnline**( )
java	boolean **isOnline**( )
py	**isOnline**( )
php	function **isOnline**( )
es	async **isOnline**( )
dnp	bool **isOnline**( )
cp	bool **isOnline**( )

If there is a cached value for the serial port 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 serial port.

Returns :
`true` if the serial port can be reached, and `false` otherwise

serialport→isOnline_async() YSerialPort

Checks if the serial port is currently reachable, without raising any error (asynchronous version).

```
js function isOnline_async( callback, context)
```

If there is a cached value for the serial port 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.

serialport→isReadOnly() YSerialPort

Test if the function is readOnly.

cpp	bool **isReadOnly**( )
m	-(bool) **isReadOnly**
pas	boolean **isReadOnly**( ): boolean
vb	function **isReadOnly**( ) As Boolean
cs	bool **isReadOnly**( )
java	boolean **isReadOnly**( )
uwp	async Task<bool> **isReadOnly**( )
py	**isReadOnly**( )
php	function **isReadOnly**( )
es	async **isReadOnly**( )
dnp	bool **isReadOnly**( )
cp	bool **isReadOnly**( )
cmd	YSerialPort **target isReadOnly**

Return `true` if the function is write protected or that the function is not available.

Returns :
`true` if the function is readOnly or not online.

serialport→load() YSerialPort

Preloads the serial port cache with a specified validity duration.

```
js    function load( msValidity)
cpp   YRETCODE load( int msValidity)
m     -(YRETCODE) load : (u64) msValidity
pas   YRETCODE load( msValidity: u64): YRETCODE
vb    function load( ByVal msValidity As Long) As YRETCODE
cs    YRETCODE load( ulong msValidity)
java  int load( long msValidity)
py    load( msValidity)
php   function load( $msValidity)
es    async load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→loadAttribute() YSerialPort

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
js    function loadAttribute( attrName)
cpp   string loadAttribute( string attrName)
m     -(NSString*) loadAttribute : (NSString*) attrName
pas   string loadAttribute( attrName: string): string
vb    function loadAttribute( ByVal attrName As String) As String
cs    string loadAttribute( string attrName)
java  String loadAttribute( String attrName)
uwp   async Task<string> loadAttribute( string attrName)
py    loadAttribute( attrName)
php   function loadAttribute( $attrName)
es    async loadAttribute( attrName)
dnp   string loadAttribute( string attrName)
cp    string loadAttribute( string attrName)
```

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

serialport→load_async() YSerialPort

Preloads the serial port cache with a specified validity duration (asynchronous version).

```
js function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking the JavaScript virtual machine.

Parameters :

msValidity an integer corresponding to the validity of the loaded function parameters, in milliseconds

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

serialport→modbusReadBits() YSerialPort

Reads one or more contiguous internal bits (or coil status) from a MODBUS serial device.

```
js   function modbusReadBits( slaveNo, pduAddr, nBits)
cpp  vector<int> modbusReadBits( int slaveNo, int pduAddr, int nBits)
m    -(NSMutableArray*) modbusReadBits : (int) slaveNo
                                       : (int) pduAddr
                                       : (int) nBits
pas  TLongIntArray modbusReadBits( slaveNo: LongInt,
                                   pduAddr: LongInt,
                                   nBits: LongInt): TLongIntArray
vb   function modbusReadBits( ByVal slaveNo As Integer,
                              ByVal pduAddr As Integer,
                              ByVal nBits As Integer) As List
cs   List<int> modbusReadBits( int slaveNo, int pduAddr, int nBits)
java ArrayList<Integer> modbusReadBits( int slaveNo,
                                        int pduAddr,
                                        int nBits)
uwp  async Task<List<int>> modbusReadBits( int slaveNo,
                                           int pduAddr,
                                           int nBits)
py   modbusReadBits( slaveNo, pduAddr, nBits)
php  function modbusReadBits( $slaveNo, $pduAddr, $nBits)
es   async modbusReadBits( slaveNo, pduAddr, nBits)
dnp  int[] modbusReadBits( int slaveNo, int pduAddr, int nBits)
cp   vector<int> modbusReadBits( int slaveNo,
                                 int pduAddr,
                                 int nBits)
cmd  YSerialPort target modbusReadBits slaveNo pduAddr nBits
```

This method uses the MODBUS function code 0x01 (Read Coils).

Parameters :

slaveNo the address of the slave MODBUS device to query
pduAddr the relative address of the first bit/coil to read (zero-based)
nBits the number of bits/coils to read

Returns :

a vector of integers, each corresponding to one bit.

On failure, throws an exception or returns an empty array.

serialport→modbusReadInputBits() YSerialPort

Reads one or more contiguous input bits (or discrete inputs) from a MODBUS serial device.

```
js   function modbusReadInputBits( slaveNo, pduAddr, nBits)
cpp  vector<int> modbusReadInputBits( int slaveNo, int pduAddr, int nBits)
m    -(NSMutableArray*) modbusReadInputBits : (int) slaveNo
                                            : (int) pduAddr
                                            : (int) nBits
pas  TLongIntArray modbusReadInputBits( slaveNo: LongInt,
                                        pduAddr: LongInt,
                                        nBits: LongInt): TLongIntArray
vb   function modbusReadInputBits( ByVal slaveNo As Integer,
                                   ByVal pduAddr As Integer,
                                   ByVal nBits As Integer) As List
cs   List<int> modbusReadInputBits( int slaveNo,
                                    int pduAddr,
                                    int nBits)
java ArrayList<Integer> modbusReadInputBits( int slaveNo,
                                             int pduAddr,
                                             int nBits)
uwp  async Task<List<int>> modbusReadInputBits( int slaveNo,
                                                int pduAddr,
                                                int nBits)
py   modbusReadInputBits( slaveNo, pduAddr, nBits)
php  function modbusReadInputBits( $slaveNo, $pduAddr, $nBits)
es   async modbusReadInputBits( slaveNo, pduAddr, nBits)
dnp  int[] modbusReadInputBits( int slaveNo, int pduAddr, int nBits)
cp   vector<int> modbusReadInputBits( int slaveNo,
                                      int pduAddr,
                                      int nBits)
cmd  YSerialPort target modbusReadInputBits slaveNo pduAddr nBits
```

This method uses the MODBUS function code 0x02 (Read Discrete Inputs).

Parameters :

slaveNo the address of the slave MODBUS device to query

pduAddr the relative address of the first bit/input to read (zero-based)

nBits the number of bits/inputs to read

Returns :

a vector of integers, each corresponding to one bit.

On failure, throws an exception or returns an empty array.

serialport→modbusReadInputRegisters() YSerialPort

Reads one or more contiguous input registers (read-only registers) from a MODBUS serial device.

```
js    function modbusReadInputRegisters( slaveNo, pduAddr, nWords)
cpp   vector<int> modbusReadInputRegisters( int slaveNo, int pduAddr, int nWords)
m     -(NSMutableArray*) modbusReadInputRegisters : (int) slaveNo
                                                  : (int) pduAddr
                                                  : (int) nWords
pas   TLongIntArray modbusReadInputRegisters( slaveNo: LongInt,
                                              pduAddr: LongInt,
                                              nWords: LongInt): TLongIntArray
vb    function modbusReadInputRegisters( ByVal slaveNo As Integer,
                                         ByVal pduAddr As Integer,
                                         ByVal nWords As Integer) As List
cs    List<int> modbusReadInputRegisters( int slaveNo,
                                          int pduAddr,
                                          int nWords)
java  ArrayList<Integer> modbusReadInputRegisters( int slaveNo,
                                                   int pduAddr,
                                                   int nWords)
uwp   async Task<List<int>> modbusReadInputRegisters( int slaveNo,
                                                      int pduAddr,
                                                      int nWords)
py    modbusReadInputRegisters( slaveNo, pduAddr, nWords)
php   function modbusReadInputRegisters( $slaveNo, $pduAddr, $nWords)
es    async modbusReadInputRegisters( slaveNo, pduAddr, nWords)
dnp   int[] modbusReadInputRegisters( int slaveNo,
                                      int pduAddr,
                                      int nWords)
cp    vector<int> modbusReadInputRegisters( int slaveNo,
                                            int pduAddr,
                                            int nWords)
cmd   YSerialPort target modbusReadInputRegisters slaveNo pduAddr nWords
```

This method uses the MODBUS function code 0x04 (Read Input Registers).

Parameters :

- **slaveNo** the address of the slave MODBUS device to query
- **pduAddr** the relative address of the first input register to read (zero-based)
- **nWords** the number of input registers to read

Returns :

a vector of integers, each corresponding to one 16-bit input value.

On failure, throws an exception or returns an empty array.

serialport→modbusReadRegisters() YSerialPort

Reads one or more contiguous internal registers (holding registers) from a MODBUS serial device.

```
js    function modbusReadRegisters( slaveNo, pduAddr, nWords)
cpp   vector<int> modbusReadRegisters( int slaveNo, int pduAddr, int nWords)
m     -(NSMutableArray*) modbusReadRegisters : (int) slaveNo
                                             : (int) pduAddr
                                             : (int) nWords
pas   TLongIntArray modbusReadRegisters( slaveNo: LongInt,
                                         pduAddr: LongInt,
                                         nWords: LongInt): TLongIntArray
vb    function modbusReadRegisters( ByVal slaveNo As Integer,
                                    ByVal pduAddr As Integer,
                                    ByVal nWords As Integer) As List
cs    List<int> modbusReadRegisters( int slaveNo,
                                     int pduAddr,
                                     int nWords)
java  ArrayList<Integer> modbusReadRegisters( int slaveNo,
                                              int pduAddr,
                                              int nWords)
uwp   async Task<List<int>> modbusReadRegisters( int slaveNo,
                                                 int pduAddr,
                                                 int nWords)
py    modbusReadRegisters( slaveNo, pduAddr, nWords)
php   function modbusReadRegisters( $slaveNo, $pduAddr, $nWords)
es    async modbusReadRegisters( slaveNo, pduAddr, nWords)
dnp   int[] modbusReadRegisters( int slaveNo,
                                 int pduAddr,
                                 int nWords)
cp    vector<int> modbusReadRegisters( int slaveNo,
                                       int pduAddr,
                                       int nWords)
cmd   YSerialPort target modbusReadRegisters slaveNo pduAddr nWords
```

This method uses the MODBUS function code 0x03 (Read Holding Registers).

Parameters :

slaveNo the address of the slave MODBUS device to query
pduAddr the relative address of the first holding register to read (zero-based)
nWords the number of holding registers to read

Returns :

a vector of integers, each corresponding to one 16-bit register value.

On failure, throws an exception or returns an empty array.

serialport→modbusWriteAndReadRegisters() YSerialPort

Sets several contiguous internal registers (holding registers) on a MODBUS serial device, then performs a contiguous read of a set of (possibly different) internal registers.

```
js    function modbusWriteAndReadRegisters( slaveNo, pduWriteAddr, values, pduReadAddr,
                                            nReadWords)
cpp   vector<int> modbusWriteAndReadRegisters( int slaveNo,
                                               int pduWriteAddr,
                                               vector<int> values,
                                               int pduReadAddr,
                                               int nReadWords)
m     -(NSMutableArray*) modbusWriteAndReadRegisters : (int) slaveNo
                                                     : (int) pduWriteAddr
                                                     : (NSMutableArray*) values
                                                     : (int) pduReadAddr
                                                     : (int) nReadWords
pas   TLongIntArray modbusWriteAndReadRegisters( slaveNo: LongInt,
                                                 pduWriteAddr: LongInt,
                                                 values: TLongIntArray,
                                                 pduReadAddr: LongInt,
                                                 nReadWords: LongInt): TLongIntArray
vb    procedure modbusWriteAndReadRegisters( ByVal slaveNo As Integer,
                                             ByVal pduWriteAddr As Integer,
                                             ByVal values As List(Of)
cs    List<int> modbusWriteAndReadRegisters( int slaveNo,
                                             int pduWriteAddr,
                                             List<int> values,
                                             int pduReadAddr,
                                             int nReadWords)
java  ArrayList<Integer> modbusWriteAndReadRegisters( int slaveNo,
                                                      int pduWriteAddr,
                                                      ArrayList<Integer> values,
                                                      int pduReadAddr,
                                                      int nReadWords)
uwp   async Task<List<int>> modbusWriteAndReadRegisters( int slaveNo,
                                                         int pduWriteAddr,
                                                         List<int> values,
                                                         int pduReadAddr,
                                                         int nReadWords)
py    modbusWriteAndReadRegisters( slaveNo, pduWriteAddr, values, pduReadAddr, nReadWords)
php   function modbusWriteAndReadRegisters( $slaveNo, $pduWriteAddr, $values, $pduReadAddr,
                                            $nReadWords)
es    async modbusWriteAndReadRegisters( slaveNo, pduWriteAddr, values, pduReadAddr,
                                         nReadWords)
dnp   int[] modbusWriteAndReadRegisters( int slaveNo,
                                         int pduWriteAddr,
                                         int[] values,
                                         int pduReadAddr,
                                         int nReadWords)
cp    vector<int> modbusWriteAndReadRegisters( int slaveNo,
```

```
                                                    int pduWriteAddr,
                                                    vector<int> values,
                                                    int pduReadAddr,
                                                    int nReadWords)
cmd  YSerialPort target modbusWriteAndReadRegisters slaveNo pduWriteAddr values pduReadAddr
                                                    nReadWords
```

This method uses the MODBUS function code 0x17 (Read/Write Multiple Registers).

Parameters :

slaveNo	the address of the slave MODBUS device to drive
pduWriteAddr	the relative address of the first internal register to set (zero-based)
values	the vector of 16 bit values to set
pduReadAddr	the relative address of the first internal register to read (zero-based)
nReadWords	the number of 16 bit values to read

Returns :

a vector of integers, each corresponding to one 16-bit register value read.

On failure, throws an exception or returns an empty array.

serialport→modbusWriteBit() YSerialPort

Sets a single internal bit (or coil) on a MODBUS serial device.

```
js    function modbusWriteBit( slaveNo, pduAddr, value)
cpp   int modbusWriteBit( int slaveNo, int pduAddr, int value)
m     -(int) modbusWriteBit : (int) slaveNo
                            : (int) pduAddr
                            : (int) value
pas   LongInt modbusWriteBit( slaveNo: LongInt,
                              pduAddr: LongInt,
                              value: LongInt): LongInt
vb    function modbusWriteBit( ByVal slaveNo As Integer,
                               ByVal pduAddr As Integer,
                               ByVal value As Integer) As Integer
cs    int modbusWriteBit( int slaveNo, int pduAddr, int value)
java  int modbusWriteBit( int slaveNo, int pduAddr, int value)
uwp   async Task<int> modbusWriteBit( int slaveNo, int pduAddr, int value)
py    modbusWriteBit( slaveNo, pduAddr, value)
php   function modbusWriteBit( $slaveNo, $pduAddr, $value)
es    async modbusWriteBit( slaveNo, pduAddr, value)
dnp   int modbusWriteBit( int slaveNo, int pduAddr, int value)
cp    int modbusWriteBit( int slaveNo, int pduAddr, int value)
cmd   YSerialPort target modbusWriteBit slaveNo pduAddr value
```

This method uses the MODBUS function code 0x05 (Write Single Coil).

Parameters :

slaveNo	the address of the slave MODBUS device to drive
pduAddr	the relative address of the bit/coil to set (zero-based)
value	the value to set (0 for OFF state, non-zero for ON state)

Returns :

the number of bits/coils affected on the device (1)

On failure, throws an exception or returns zero.

serialport→modbusWriteBits() YSerialPort

Sets several contiguous internal bits (or coils) on a MODBUS serial device.

```
js   function modbusWriteBits( slaveNo, pduAddr, bits)
cpp  int modbusWriteBits( int slaveNo, int pduAddr, vector<int> bits)
m    -(int) modbusWriteBits : (int) slaveNo
                            : (int) pduAddr
                            : (NSMutableArray*) bits
pas  LongInt modbusWriteBits( slaveNo: LongInt,
                              pduAddr: LongInt,
                              bits: TLongIntArray): LongInt
vb   procedure modbusWriteBits( ByVal slaveNo As Integer,
                                ByVal pduAddr As Integer,
                                ByVal bits As List(Of)
cs   int modbusWriteBits( int slaveNo, int pduAddr, List<int> bits)
java int modbusWriteBits( int slaveNo,
                          int pduAddr,
                          ArrayList<Integer> bits)
uwp  async Task<int> modbusWriteBits( int slaveNo,
                                      int pduAddr,
                                      List<int> bits)
py   modbusWriteBits( slaveNo, pduAddr, bits)
php  function modbusWriteBits( $slaveNo, $pduAddr, $bits)
es   async modbusWriteBits( slaveNo, pduAddr, bits)
dnp  int modbusWriteBits( int slaveNo, int pduAddr, int[] bits)
cp   int modbusWriteBits( int slaveNo,
                          int pduAddr,
                          vector<int> bits)
cmd  YSerialPort target modbusWriteBits slaveNo pduAddr bits
```

This method uses the MODBUS function code 0x0f (Write Multiple Coils).

Parameters :

slaveNo	the address of the slave MODBUS device to drive
pduAddr	the relative address of the first bit/coil to set (zero-based)
bits	the vector of bits to be set (one integer per bit)

Returns :

the number of bits/coils affected on the device

On failure, throws an exception or returns zero.

serialport→modbusWriteRegister() YSerialPort

Sets a single internal register (or holding register) on a MODBUS serial device.

```
js   function modbusWriteRegister( slaveNo, pduAddr, value)
cpp  int modbusWriteRegister( int slaveNo, int pduAddr, int value)
m    -(int) modbusWriteRegister : (int) slaveNo
                                : (int) pduAddr
                                : (int) value
pas  LongInt modbusWriteRegister( slaveNo: LongInt,
                                  pduAddr: LongInt,
                                  value: LongInt): LongInt
vb   function modbusWriteRegister( ByVal slaveNo As Integer,
                                   ByVal pduAddr As Integer,
                                   ByVal value As Integer) As Integer
cs   int modbusWriteRegister( int slaveNo, int pduAddr, int value)
java int modbusWriteRegister( int slaveNo, int pduAddr, int value)
uwp  async Task<int> modbusWriteRegister( int slaveNo,
                                          int pduAddr,
                                          int value)
py   modbusWriteRegister( slaveNo, pduAddr, value)
php  function modbusWriteRegister( $slaveNo, $pduAddr, $value)
es   async modbusWriteRegister( slaveNo, pduAddr, value)
dnp  int modbusWriteRegister( int slaveNo, int pduAddr, int value)
cp   int modbusWriteRegister( int slaveNo, int pduAddr, int value)
cmd  YSerialPort target modbusWriteRegister slaveNo pduAddr value
```

This method uses the MODBUS function code 0x06 (Write Single Register).

Parameters :

- **slaveNo** the address of the slave MODBUS device to drive
- **pduAddr** the relative address of the register to set (zero-based)
- **value** the 16 bit value to set

Returns :

the number of registers affected on the device (1)

On failure, throws an exception or returns zero.

serialport→modbusWriteRegisters() YSerialPort

Sets several contiguous internal registers (or holding registers) on a MODBUS serial device.

```
js    function modbusWriteRegisters( slaveNo, pduAddr, values)
cpp   int modbusWriteRegisters( int slaveNo, int pduAddr, vector<int> values)
m     -(int) modbusWriteRegisters : (int) slaveNo
                                  : (int) pduAddr
                                  : (NSMutableArray*) values
pas   LongInt modbusWriteRegisters( slaveNo: LongInt,
                                    pduAddr: LongInt,
                                    values: TLongIntArray): LongInt
vb    procedure modbusWriteRegisters( ByVal slaveNo As Integer,
                                      ByVal pduAddr As Integer,
                                      ByVal values As List(Of)
cs    int modbusWriteRegisters( int slaveNo,
                                int pduAddr,
                                List<int> values)
java  int modbusWriteRegisters( int slaveNo,
                                int pduAddr,
                                ArrayList<Integer> values)
uwp   async Task<int> modbusWriteRegisters( int slaveNo,
                                            int pduAddr,
                                            List<int> values)
py    modbusWriteRegisters( slaveNo, pduAddr, values)
php   function modbusWriteRegisters( $slaveNo, $pduAddr, $values)
es    async modbusWriteRegisters( slaveNo, pduAddr, values)
dnp   int modbusWriteRegisters( int slaveNo,
                                int pduAddr,
                                int[] values)
cp    int modbusWriteRegisters( int slaveNo,
                                int pduAddr,
                                vector<int> values)
cmd   YSerialPort target modbusWriteRegisters slaveNo pduAddr values
```

This method uses the MODBUS function code 0x10 (Write Multiple Registers).

Parameters :

slaveNo	the address of the slave MODBUS device to drive
pduAddr	the relative address of the first internal register to set (zero-based)
values	the vector of 16 bit values to set

Returns :

the number of registers affected on the device

On failure, throws an exception or returns zero.

serialport→muteValueCallbacks() YSerialPort

Disables the propagation of every new advertised value to the parent hub.

js	function **muteValueCallbacks**( )
cpp	int **muteValueCallbacks**( )
m	-(int) **muteValueCallbacks**
pas	LongInt **muteValueCallbacks**( ): LongInt
vb	function **muteValueCallbacks**( ) As Integer
cs	int **muteValueCallbacks**( )
java	int **muteValueCallbacks**( )
uwp	async Task<int> **muteValueCallbacks**( )
py	**muteValueCallbacks**( )
php	function **muteValueCallbacks**( )
es	async **muteValueCallbacks**( )
dnp	int **muteValueCallbacks**( )
cp	int **muteValueCallbacks**( )
cmd	YSerialPort **target muteValueCallbacks**

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→nextSerialPort() YSerialPort

Continues the enumeration of serial ports started using `yFirstSerialPort()`.

js	function **nextSerialPort**( )
cpp	YSerialPort * **nextSerialPort**( )
m	-(YSerialPort*) **nextSerialPort**
pas	TYSerialPort **nextSerialPort**( ): TYSerialPort
vb	function **nextSerialPort**( ) As YSerialPort
cs	YSerialPort **nextSerialPort**( )
java	YSerialPort **nextSerialPort**( )
uwp	YSerialPort **nextSerialPort**( )
py	**nextSerialPort**( )
php	function **nextSerialPort**( )
es	**nextSerialPort**( )

Caution: You can't make any assumption about the returned serial ports order. If you want to find a specific a serial port, use `SerialPort.findSerialPort()` and a hardwareID or a logical name.

Returns :

a pointer to a `YSerialPort` object, corresponding to a serial port currently online, or a `null` pointer if there are no more serial ports to enumerate.

serialport→queryHex() YSerialPort

Sends a binary message to the serial port, and reads the reply, if any.

```
js    function queryHex( hexString, maxWait)
cpp   string queryHex( string hexString, int maxWait)
m     -(NSString*) queryHex : (NSString*) hexString
                            : (int) maxWait
pas   string queryHex( hexString: string, maxWait: LongInt): string
vb    function queryHex( ByVal hexString As String, ByVal maxWait As Integer) As String
cs    string queryHex( string hexString, int maxWait)
java  String queryHex( String hexString, int maxWait)
uwp   async Task<string> queryHex( string hexString, int maxWait)
py    queryHex( hexString, maxWait)
php   function queryHex( $hexString, $maxWait)
es    async queryHex( hexString, maxWait)
dnp   string queryHex( string hexString, int maxWait)
cp    string queryHex( string hexString, int maxWait)
cmd   YSerialPort target queryHex hexString maxWait
```

This function is intended to be used when the serial port is configured for Frame-based protocol.

Parameters :

hexString the message to send, coded in hexadecimal

maxWait the maximum number of milliseconds to wait for a reply.

Returns :

the next frame received after sending the message, as a hex string. Additional frames can be obtained by calling readHex or readMessages.

On failure, throws an exception or returns an empty string.

serialport→queryLine() YSerialPort

Sends a text line query to the serial port, and reads the reply, if any.

```
js    function queryLine( query, maxWait)
cpp   string queryLine( string query, int maxWait)
m     -(NSString*) queryLine : (NSString*) query : (int) maxWait
pas   string queryLine( query: string, maxWait: LongInt): string
vb    function queryLine( ByVal query As String, ByVal maxWait As Integer) As String
cs    string queryLine( string query, int maxWait)
java  String queryLine( String query, int maxWait)
uwp   async Task<string> queryLine( string query, int maxWait)
py    queryLine( query, maxWait)
php   function queryLine( $query, $maxWait)
es    async queryLine( query, maxWait)
dnp   string queryLine( string query, int maxWait)
cp    string queryLine( string query, int maxWait)
cmd   YSerialPort target queryLine query maxWait
```

This function is intended to be used when the serial port is configured for 'Line' protocol.

Parameters :

query the line query to send (without CR/LF)

maxWait the maximum number of milliseconds to wait for a reply.

Returns :

the next text line received after sending the text query, as a string. Additional lines can be obtained by calling readLine or readMessages.

On failure, throws an exception or returns an empty string.

serialport→queryMODBUS() YSerialPort

Sends a message to a specified MODBUS slave connected to the serial port, and reads the reply, if any.

```
js    function queryMODBUS( slaveNo, pduBytes)
cpp   vector<int> queryMODBUS( int slaveNo, vector<int> pduBytes)
m     -(NSMutableArray*) queryMODBUS : (int) slaveNo
                                     : (NSMutableArray*) pduBytes
pas   TLongIntArray queryMODBUS( slaveNo: LongInt,
                                 pduBytes: TLongIntArray): TLongIntArray
vb    procedure queryMODBUS( ByVal slaveNo As Integer, ByVal pduBytes As List(Of)
cs    List<int> queryMODBUS( int slaveNo, List<int> pduBytes)
java  ArrayList<Integer> queryMODBUS( int slaveNo,
                                      ArrayList<Integer> pduBytes)
uwp   async Task<List<int>> queryMODBUS( int slaveNo, List<int> pduBytes)
py    queryMODBUS( slaveNo, pduBytes)
php   function queryMODBUS( $slaveNo, $pduBytes)
es    async queryMODBUS( slaveNo, pduBytes)
dnp   int[] queryMODBUS( int slaveNo, int[] pduBytes)
cp    vector<int> queryMODBUS( int slaveNo, vector<int> pduBytes)
cmd   YSerialPort target queryMODBUS slaveNo pduBytes
```

The message is the PDU, provided as a vector of bytes.

Parameters :

slaveNo the address of the slave MODBUS device to query

pduBytes the message to send (PDU), as a vector of bytes. The first byte of the PDU is the MODBUS function code.

Returns :

the received reply, as a vector of bytes.

On failure, throws an exception or returns an empty array (or a MODBUS error reply).

serialport→readArray() YSerialPort

Reads data from the receive buffer as a list of bytes, starting at current stream position.

```
js    function readArray( nChars)
cpp   vector<int> readArray( int nChars)
m     -(NSMutableArray*) readArray : (int) nChars
pas   TLongIntArray readArray( nChars: LongInt): TLongIntArray
vb    function readArray( ByVal nChars As Integer) As List
cs    List<int> readArray( int nChars)
java  ArrayList<Integer> readArray( int nChars)
uwp   async Task<List<int>> readArray( int nChars)
py    readArray( nChars)
php   function readArray( $nChars)
es    async readArray( nChars)
dnp   int[] readArray( int nChars)
cp    vector<int> readArray( int nChars)
cmd   YSerialPort target readArray nChars
```

If data at current stream position is not available anymore in the receive buffer, the function performs a short read.

Parameters :

nChars the maximum number of bytes to read

Returns :

a sequence of bytes with receive buffer contents

On failure, throws an exception or returns an empty array.

serialport→readBin() YSerialPort

Reads data from the receive buffer as a binary buffer, starting at current stream position.

js	function **readBin**(**nChars**)
cpp	string **readBin**(int **nChars**)
m	-(NSMutableData*) **readBin** : (int) **nChars**
pas	TByteArray **readBin**(**nChars**: LongInt): TByteArray
vb	function **readBin**(ByVal **nChars** As Integer) As Byte
cs	byte[] **readBin**(int **nChars**)
java	byte[] **readBin**(int **nChars**)
uwp	async Task<byte[]> **readBin**(int **nChars**)
py	**readBin**(**nChars**)
php	function **readBin**(**$nChars**)
es	async **readBin**(**nChars**)
dnp	byte[] **readBin**(int **nChars**)
cp	string **readBin**(int **nChars**)
cmd	YSerialPort **target readBin nChars**

If data at current stream position is not available anymore in the receive buffer, the function performs a short read.

Parameters :

nChars the maximum number of bytes to read

Returns :

a binary object with receive buffer contents

On failure, throws an exception or returns a negative error code.

serialport→readByte() YSerialPort

Reads one byte from the receive buffer, starting at current stream position.

js	function **readByte**( )
cpp	int **readByte**( )
m	-(int) **readByte**
pas	LongInt **readByte**( ): LongInt
vb	function **readByte**( ) As Integer
cs	int **readByte**( )
java	int **readByte**( )
uwp	async Task<int> **readByte**( )
py	**readByte**( )
php	function **readByte**( )
es	async **readByte**( )
dnp	int **readByte**( )
cp	int **readByte**( )
cmd	YSerialPort **target readByte**

If data at current stream position is not available anymore in the receive buffer, or if there is no data available yet, the function returns YAPI_NO_MORE_DATA.

Returns :
the next byte

On failure, throws an exception or returns a negative error code.

serialport→readHex() YSerialPort

Reads data from the receive buffer as a hexadecimal string, starting at current stream position.

```
js    function readHex( nBytes)
cpp   string readHex( int nBytes)
m     -(NSString*) readHex : (int) nBytes
pas   string readHex( nBytes: LongInt): string
vb    function readHex( ByVal nBytes As Integer) As String
cs    string readHex( int nBytes)
java  String readHex( int nBytes)
uwp   async Task<string> readHex( int nBytes)
py    readHex( nBytes)
php   function readHex( $nBytes)
es    async readHex( nBytes)
dnp   string readHex( int nBytes)
cp    string readHex( int nBytes)
cmd   YSerialPort target readHex nBytes
```

If data at current stream position is not available anymore in the receive buffer, the function performs a short read.

Parameters :

nBytes the maximum number of bytes to read

Returns :

a string with receive buffer contents, encoded in hexadecimal

On failure, throws an exception or returns a negative error code.

serialport→readLine() YSerialPort

Reads a single line (or message) from the receive buffer, starting at current stream position.

js	function **readLine**( )
cpp	string **readLine**( )
m	-(NSString*) **readLine**
pas	string **readLine**( ): string
vb	function **readLine**( ) As String
cs	string **readLine**( )
java	String **readLine**( )
uwp	async Task<string> **readLine**( )
py	**readLine**( )
php	function **readLine**( )
es	async **readLine**( )
dnp	string **readLine**( )
cp	string **readLine**( )
cmd	YSerialPort **target readLine**

This function is intended to be used when the serial port is configured for a message protocol, such as 'Line' mode or frame protocols.

If data at current stream position is not available anymore in the receive buffer, the function returns the oldest available line and moves the stream position just after. If no new full line is received, the function returns an empty line.

Returns :
a string with a single line of text

On failure, throws an exception or returns a negative error code.

serialport→readMessages() YSerialPort

Searches for incoming messages in the serial port receive buffer matching a given pattern, starting at current position.

```
js    function readMessages( pattern, maxWait)
cpp   vector<string> readMessages( string pattern, int maxWait)
m     -(NSMutableArray*) readMessages : (NSString*) pattern
                                      : (int) maxWait
pas   TStringArray readMessages( pattern: string, maxWait: LongInt): TStringArray
vb    function readMessages( ByVal pattern As String, ByVal maxWait As Integer) As List
cs    List<string> readMessages( string pattern, int maxWait)
java  ArrayList<String> readMessages( String pattern, int maxWait)
uwp   async Task<List<string>> readMessages( string pattern, int maxWait)
py    readMessages( pattern, maxWait)
php   function readMessages( $pattern, $maxWait)
es    async readMessages( pattern, maxWait)
dnp   string[] readMessages( string pattern, int maxWait)
cp    vector<string> readMessages( string pattern, int maxWait)
cmd   YSerialPort target readMessages pattern maxWait
```

This function will only compare and return printable characters in the message strings. Binary protocols are handled as hexadecimal strings.

The search returns all messages matching the expression provided as argument in the buffer. If no matching message is found, the search waits for one up to the specified maximum timeout (in milliseconds).

Parameters :

pattern a limited regular expression describing the expected message format, or an empty string if all messages should be returned (no filtering). When using binary protocols, the format applies to the hexadecimal representation of the message.

maxWait the maximum number of milliseconds to wait for a message if none is found in the receive buffer.

Returns :

an array of strings containing the messages found, if any. Binary messages are converted to hexadecimal representation.

On failure, throws an exception or returns an empty array.

serialport→readStr() YSerialPort

Reads data from the receive buffer as a string, starting at current stream position.

```
js    function readStr( nChars)
cpp   string readStr( int nChars)
m     -(NSString*) readStr : (int) nChars
pas   string readStr( nChars: LongInt): string
vb    function readStr( ByVal nChars As Integer) As String
cs    string readStr( int nChars)
java  String readStr( int nChars)
uwp   async Task<string> readStr( int nChars)
py    readStr( nChars)
php   function readStr( $nChars)
es    async readStr( nChars)
dnp   string readStr( int nChars)
cp    string readStr( int nChars)
cmd   YSerialPort target readStr nChars
```

If data at current stream position is not available anymore in the receive buffer, the function performs a short read.

Parameters :

nChars the maximum number of characters to read

Returns :

a string with receive buffer contents

On failure, throws an exception or returns a negative error code.

serialport→read_avail() YSerialPort

Returns the number of bytes available to read in the input buffer starting from the current absolute stream position pointer of the API object.

```
js    function read_avail( )
cpp   int read_avail( )
m     -(int) read_avail
pas   LongInt read_avail( ): LongInt
vb    function read_avail( ) As Integer
cs    int read_avail( )
java  int read_avail( )
uwp   async Task<int> read_avail( )
py    read_avail( )
php   function read_avail( )
es    async read_avail( )
dnp   int read_avail( )
cp    int read_avail( )
cmd   YSerialPort target read_avail
```

Returns :
the number of bytes available to read

serialport→read_seek() YSerialPort

Changes the current internal stream position to the specified value.

js	function **read_seek**(**absPos**)
cpp	int **read_seek**(int **absPos**)
m	-(int) **read_seek** : (int) **absPos**
pas	LongInt **read_seek**(**absPos**: LongInt): LongInt
vb	function **read_seek**(ByVal **absPos** As Integer) As Integer
cs	int **read_seek**(int **absPos**)
java	int **read_seek**(int **absPos**)
uwp	async Task<int> **read_seek**(int **absPos**)
py	**read_seek**(**absPos**)
php	function **read_seek**(**$absPos**)
es	async **read_seek**(**absPos**)
dnp	int **read_seek**(int **absPos**)
cp	int **read_seek**(int **absPos**)
cmd	YSerialPort **target read_seek absPos**

This function does not affect the device, it only changes the value stored in the API object for the next read operations.

Parameters :

absPos the absolute position index for next read operations.

Returns :

nothing.

serialport→read_tell() YSerialPort

Returns the current absolute stream position pointer of the API object.

js	function **read_tell**( )
cpp	int **read_tell**( )
m	-(int) **read_tell**
pas	LongInt **read_tell**( ): LongInt
vb	function **read_tell**( ) As Integer
cs	int **read_tell**( )
java	int **read_tell**( )
uwp	async Task<int> **read_tell**( )
py	**read_tell**( )
php	function **read_tell**( )
es	async **read_tell**( )
dnp	int **read_tell**( )
cp	int **read_tell**( )
cmd	YSerialPort **target read_tell**

Returns :
the absolute position index for next read operations.

serialport→registerValueCallback() YSerialPort

Registers the callback function that is invoked on every change of advertised value.

js	function **registerValueCallback**(**callback**)
cpp	int **registerValueCallback**(YSerialPortValueCallback **callback**)
m	-(int) **registerValueCallback** : (YSerialPortValueCallback) **callback**
pas	LongInt **registerValueCallback**(**callback**: TYSerialPortValueCallback): LongInt
vb	function **registerValueCallback**(ByVal **callback** As YSerialPortValueCallback) As Integer
cs	int **registerValueCallback**(ValueCallback **callback**)
java	int **registerValueCallback**(UpdateCallback **callback**)
uwp	async Task<int> **registerValueCallback**(ValueCallback **callback**)
py	**registerValueCallback**(**callback**)
php	function **registerValueCallback**(**$callback**)
es	async **registerValueCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

serialport→reset() YSerialPort

Clears the serial port buffer and resets counters to zero.

js	function **reset**( )
cpp	int **reset**( )
m	-(int) **reset**
pas	LongInt **reset**( ): LongInt
vb	function **reset**( ) As Integer
cs	int **reset**( )
java	int **reset**( )
uwp	async Task<int> **reset**( )
py	**reset**( )
php	function **reset**( )
es	async **reset**( )
dnp	int **reset**( )
cp	int **reset**( )
cmd	YSerialPort **target reset**

Returns :
`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→selectJob() YSerialPort

Load and start processing the specified job file.

```
js    function selectJob( jobfile)
cpp   int selectJob( string jobfile)
m     -(int) selectJob : (NSString*) jobfile
pas   LongInt selectJob( jobfile: string): LongInt
vb    function selectJob( ByVal jobfile As String) As Integer
cs    int selectJob( string jobfile)
java  int selectJob( String jobfile)
uwp   async Task<int> selectJob( string jobfile)
py    selectJob( jobfile)
php   function selectJob( $jobfile)
es    async selectJob( jobfile)
dnp   int selectJob( string jobfile)
cp    int selectJob( string jobfile)
cmd   YSerialPort target selectJob jobfile
```

The file must have been previously created using the user interface or uploaded on the device filesystem using the `uploadJob()` function.

Parameters :

jobfile name of the job file (on the device filesystem)

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→set_RTS() YSerialPort
serialport→setRTS()

Manually sets the state of the RTS line.

```
js    function set_RTS( val)
cpp   int set_RTS( int val)
m     -(int) setRTS : (int) val
pas   LongInt set_RTS( val: LongInt): LongInt
vb    function set_RTS( ByVal val As Integer) As Integer
cs    int set_RTS( int val)
java  int set_RTS( int val)
uwp   async Task<int> set_RTS( int val)
py    set_RTS( val)
php   function set_RTS( $val)
es    async set_RTS( val)
dnp   int set_RTS( int val)
cp    int set_RTS( int val)
cmd   YSerialPort target set_RTS val
```

This function has no effect when hardware handshake is enabled, as the RTS line is driven automatically.

Parameters :

val 1 to turn RTS on, 0 to turn RTS off

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→set_currentJob() YSerialPort
serialport→setCurrentJob()

Selects a job file to run immediately.

```
js    function set_currentJob( newval)
cpp   int set_currentJob( string newval)
m     -(int) setCurrentJob : (NSString*) newval
pas   integer set_currentJob( newval: string): integer
vb    function set_currentJob( ByVal newval As String) As Integer
cs    int set_currentJob( string newval)
java  int set_currentJob( String newval)
uwp   async Task<int> set_currentJob( string newval)
py    set_currentJob( newval)
php   function set_currentJob( $newval)
es    async set_currentJob( newval)
dnp   int set_currentJob( string newval)
cp    int set_currentJob( string newval)
cmd   YSerialPort target set_currentJob newval
```

If an empty string is given as argument, stops running current job file.

Parameters :

newval a string

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→set_logicalName() YSerialPort
serialport→setLogicalName()

Changes the logical name of the serial port.

```
js    function set_logicalName( newval)
cpp   int set_logicalName( string newval)
m     -(int) setLogicalName : (NSString*) newval
pas   integer set_logicalName( newval: string): integer
vb    function set_logicalName( ByVal newval As String) As Integer
cs    int set_logicalName( string newval)
java  int set_logicalName( String newval)
uwp   async Task<int> set_logicalName( string newval)
py    set_logicalName( newval)
php   function set_logicalName( $newval)
es    async set_logicalName( newval)
dnp   int set_logicalName( string newval)
cp    int set_logicalName( string newval)
cmd   YSerialPort target set_logicalName newval
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the serial port.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→set_protocol() YSerialPort
serialport→setProtocol()

Changes the type of protocol used over the serial line.

```
js    function set_protocol( newval)
cpp   int set_protocol( string newval)
m     -(int) setProtocol : (NSString*) newval
pas   integer set_protocol( newval: string): integer
vb    function set_protocol( ByVal newval As String) As Integer
cs    int set_protocol( string newval)
java  int set_protocol( String newval)
uwp   async Task<int> set_protocol( string newval)
py    set_protocol( newval)
php   function set_protocol( $newval)
es    async set_protocol( newval)
dnp   int set_protocol( string newval)
cp    int set_protocol( string newval)
cmd   YSerialPort target set_protocol newval
```

Possible values are "Line" for ASCII messages separated by CR and/or LF, "Frame:[timeout]ms" for binary messages separated by a delay time, "Modbus-ASCII" for MODBUS messages in ASCII mode, "Modbus-RTU" for MODBUS messages in RTU mode, "Wiegand-ASCII" for Wiegand messages in ASCII mode, "Wiegand-26","Wiegand-34", etc for Wiegand messages in byte mode, "Char" for a continuous ASCII stream or "Byte" for a continuous binary stream. The suffix "/[wait]ms" can be added to reduce the transmit rate so that there is always at lest the specified number of milliseconds between each bytes sent. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the type of protocol used over the serial line

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→set_serialMode() YSerialPort
serialport→setSerialMode()

Changes the serial port communication parameters, with a string such as "9600,8N1".

```
js    function set_serialMode( newval)
cpp   int set_serialMode( string newval)
m     -(int) setSerialMode : (NSString*) newval
pas   integer set_serialMode( newval: string): integer
vb    function set_serialMode( ByVal newval As String) As Integer
cs    int set_serialMode( string newval)
java  int set_serialMode( String newval)
uwp   async Task<int> set_serialMode( string newval)
py    set_serialMode( newval)
php   function set_serialMode( $newval)
es    async set_serialMode( newval)
dnp   int set_serialMode( string newval)
cp    int set_serialMode( string newval)
cmd   YSerialPort target set_serialMode newval
```

The string includes the baud rate, the number of data bits, the parity, and the number of stop bits. An optional suffix can be added to enable flow control: "CtsRts" for hardware handshake, "XOnXOff" for logical flow control and "Simplex" for acquiring a shared bus using the RTS line (as used by some RS485 adapters for instance). Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the serial port communication parameters, with a string such as "9600,8N1"

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→set_startupJob() YSerialPort
serialport→setStartupJob()

Changes the job to use when the device is powered on.

```
js    function set_startupJob( newval)
cpp   int set_startupJob( string newval)
m     -(int) setStartupJob : (NSString*) newval
pas   integer set_startupJob( newval: string): integer
vb    function set_startupJob( ByVal newval As String) As Integer
cs    int set_startupJob( string newval)
java  int set_startupJob( String newval)
uwp   async Task<int> set_startupJob( string newval)
py    set_startupJob( newval)
php   function set_startupJob( $newval)
es    async set_startupJob( newval)
dnp   int set_startupJob( string newval)
cp    int set_startupJob( string newval)
cmd   YSerialPort target set_startupJob newval
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the job to use when the device is powered on

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→set_userData() YSerialPort
serialport→setUserData()

Stores a user context provided as argument in the userData attribute of the function.

```
js    function set_userData( data)
cpp   void set_userData( void * data)
m     -(void) setUserData : (id) data
pas   set_userData( data: Tobject)
vb    procedure set_userData( ByVal data As Object)
cs    void set_userData( object data)
java  void set_userData( Object data)
py    set_userData( data)
php   function set_userData( $data)
es    async set_userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

serialport→set_voltageLevel() YSerialPort
serialport→setVoltageLevel()

Changes the voltage type used on the serial line.

js	function **set_voltageLevel**(**newval**)
cpp	int **set_voltageLevel**(Y_VOLTAGELEVEL_enum **newval**)
m	-(int) setVoltageLevel : (Y_VOLTAGELEVEL_enum) **newval**
pas	integer **set_voltageLevel**(**newval**: Integer): integer
vb	function **set_voltageLevel**(ByVal **newval** As Integer) As Integer
cs	int **set_voltageLevel**(int **newval**)
java	int **set_voltageLevel**(int **newval**)
uwp	async Task<int> **set_voltageLevel**(int **newval**)
py	**set_voltageLevel**(**newval**)
php	function **set_voltageLevel**(**$newval**)
es	async **set_voltageLevel**(**newval**)
dnp	int **set_voltageLevel**(int **newval**)
cp	int **set_voltageLevel**(int **newval**)
cmd	YSerialPort **target set_voltageLevel newval**

Valid values will depend on the Yoctopuce device model featuring the serial port feature. Check your device documentation to find out which values are valid for that specific model. Trying to set an invalid value will have no effect. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a value among `Y_VOLTAGELEVEL_OFF`, `Y_VOLTAGELEVEL_TTL3V`, `Y_VOLTAGELEVEL_TTL3VR`, `Y_VOLTAGELEVEL_TTL5V`, `Y_VOLTAGELEVEL_TTL5VR`, `Y_VOLTAGELEVEL_RS232`, `Y_VOLTAGELEVEL_RS485` and `Y_VOLTAGELEVEL_TTL1V8` corresponding to the voltage type used on the serial line

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→snoopMessages() YSerialPort

Retrieves messages (both direction) in the serial port buffer, starting at current position.

```
js    function snoopMessages( maxWait)
cpp   vector<YSnoopingRecord> snoopMessages( int maxWait)
m     -(NSMutableArray*) snoopMessages : (int) maxWait
pas   TYSnoopingRecordArray snoopMessages( maxWait: LongInt): TYSnoopingRecordArray
vb    function snoopMessages( ByVal maxWait As Integer) As List
cs    List<YSnoopingRecord> snoopMessages( int maxWait)
java  ArrayList<YSnoopingRecord> snoopMessages( int maxWait)
uwp   async Task<List<YSnoopingRecord>> snoopMessages( int maxWait)
py    snoopMessages( maxWait)
php   function snoopMessages( $maxWait)
es    async snoopMessages( maxWait)
dnp   YSnoopingRecordProxy[] snoopMessages( int maxWait)
cp    vector<YSnoopingRecordProxy> snoopMessages( int maxWait)
cmd   YSerialPort target snoopMessages maxWait
```

This function will only compare and return printable characters in the message strings. Binary protocols are handled as hexadecimal strings.

If no message is found, the search waits for one up to the specified maximum timeout (in milliseconds).

Parameters :

maxWait the maximum number of milliseconds to wait for a message if none is found in the receive buffer.

Returns :

an array of `YSnoopingRecord` objects containing the messages found, if any. Binary messages are converted to hexadecimal representation.

On failure, throws an exception or returns an empty array.

serialport→unmuteValueCallbacks() YSerialPort

Re-enables the propagation of every new advertised value to the parent hub.

js	function **unmuteValueCallbacks**( )
cpp	int **unmuteValueCallbacks**( )
m	-(int) **unmuteValueCallbacks**
pas	LongInt **unmuteValueCallbacks**( ): LongInt
vb	function **unmuteValueCallbacks**( ) As Integer
cs	int **unmuteValueCallbacks**( )
java	int **unmuteValueCallbacks**( )
uwp	async Task<int> **unmuteValueCallbacks**( )
py	**unmuteValueCallbacks**( )
php	function **unmuteValueCallbacks**( )
es	async **unmuteValueCallbacks**( )
dnp	int **unmuteValueCallbacks**( )
cp	int **unmuteValueCallbacks**( )
cmd	YSerialPort **target unmuteValueCallbacks**

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→uploadJob() YSerialPort

Saves the job definition string (JSON data) into a job file.

```
js    function uploadJob( jobfile, jsonDef)
cpp   int uploadJob( string jobfile, string jsonDef)
m     -(int) uploadJob : (NSString*) jobfile
                       : (NSString*) jsonDef
pas   LongInt uploadJob( jobfile: string, jsonDef: string): LongInt
vb    function uploadJob( ByVal jobfile As String, ByVal jsonDef As String) As Integer
cs    int uploadJob( string jobfile, string jsonDef)
java  int uploadJob( String jobfile, String jsonDef)
uwp   async Task<int> uploadJob( string jobfile, string jsonDef)
py    uploadJob( jobfile, jsonDef)
php   function uploadJob( $jobfile, $jsonDef)
es    async uploadJob( jobfile, jsonDef)
dnp   int uploadJob( string jobfile, string jsonDef)
cp    int uploadJob( string jobfile, string jsonDef)
cmd   YSerialPort target uploadJob jobfile jsonDef
```

The job file can be later enabled using `selectJob()`.

Parameters :

jobfile name of the job file to save on the device filesystem

jsonDef a string containing a JSON definition of the job

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→wait_async() YSerialPort

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js  function wait_async( callback, context)
es  wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

serialport→writeArray() YSerialPort

Sends a byte sequence (provided as a list of bytes) to the serial port.

```
js    function writeArray( byteList)
cpp   int writeArray( vector<int> byteList)
m     -(int) writeArray : (NSMutableArray*) byteList
pas   LongInt writeArray( byteList: TLongIntArray): LongInt
vb    procedure writeArray( ByVal byteList As List(Of)
cs    int writeArray( List<int> byteList)
java  int writeArray( ArrayList<Integer> byteList)
uwp   async Task<int> writeArray( List<int> byteList)
py    writeArray( byteList)
php   function writeArray( $byteList)
es    async writeArray( byteList)
dnp   int writeArray( int[] byteList)
cp    int writeArray( vector<int> byteList)
cmd   YSerialPort target writeArray byteList
```

Parameters :

byteList a list of byte codes

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→writeBin() YSerialPort

Sends a binary buffer to the serial port, as is.

```
js    function writeBin( buff)
cpp   int writeBin( string buff)
m     -(int) writeBin : (NSData*) buff
pas   LongInt writeBin( buff: TByteArray): LongInt
vb    procedure writeBin( ByVal buff As Byte()
cs    int writeBin( byte[] buff)
java  int writeBin( byte[] buff)
uwp   async Task<int> writeBin( byte[] buff)
py    writeBin( buff)
php   function writeBin( $buff)
es    async writeBin( buff)
dnp   int writeBin( byte[] buff)
cp    int writeBin( string buff)
cmd   YSerialPort target writeBin buff
```

Parameters :

buff the binary buffer to send

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→writeByte() YSerialPort

Sends a single byte to the serial port.

js	function **writeByte**(**code**)
cpp	int **writeByte**(int **code**)
m	-(int) **writeByte** : (int) **code**
pas	LongInt **writeByte**(**code**: LongInt): LongInt
vb	function **writeByte**(ByVal **code** As Integer) As Integer
cs	int **writeByte**(int **code**)
java	int **writeByte**(int **code**)
uwp	async Task<int> **writeByte**(int **code**)
py	**writeByte**(**code**)
php	function **writeByte**(**$code**)
es	async **writeByte**(**code**)
dnp	int **writeByte**(int **code**)
cp	int **writeByte**(int **code**)
cmd	YSerialPort **target writeByte code**

Parameters :

code the byte to send

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→writeHex() YSerialPort

Sends a byte sequence (provided as a hexadecimal string) to the serial port.

js	function **writeHex**(**hexString**)
cpp	int **writeHex**(string **hexString**)
m	-(int) **writeHex** : (NSString*) **hexString**
pas	LongInt **writeHex**(**hexString**: string): LongInt
vb	function **writeHex**(ByVal **hexString** As String) As Integer
cs	int **writeHex**(string **hexString**)
java	int **writeHex**(String **hexString**)
uwp	async Task<int> **writeHex**(string **hexString**)
py	**writeHex**(**hexString**)
php	function **writeHex**(**$hexString**)
es	async **writeHex**(**hexString**)
dnp	int **writeHex**(string **hexString**)
cp	int **writeHex**(string **hexString**)
cmd	YSerialPort **target writeHex hexString**

Parameters :

hexString a string of hexadecimal byte codes

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→writeLine() YSerialPort

Sends an ASCII string to the serial port, followed by a line break (CR LF).

```
js    function writeLine( text)
cpp   int writeLine( string text)
m     -(int) writeLine : (NSString*) text
pas   LongInt writeLine( text: string): LongInt
vb    function writeLine( ByVal text As String) As Integer
cs    int writeLine( string text)
java  int writeLine( String text)
uwp   async Task<int> writeLine( string text)
py    writeLine( text)
php   function writeLine( $text)
es    async writeLine( text)
dnp   int writeLine( string text)
cp    int writeLine( string text)
cmd   YSerialPort target writeLine text
```

Parameters :

text the text string to send

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→writeMODBUS() YSerialPort

Sends a MODBUS message (provided as a hexadecimal string) to the serial port.

```
js    function writeMODBUS( hexString)
cpp   int writeMODBUS( string hexString)
m     -(int) writeMODBUS : (NSString*) hexString
pas   LongInt writeMODBUS( hexString: string): LongInt
vb    function writeMODBUS( ByVal hexString As String) As Integer
cs    int writeMODBUS( string hexString)
java  int writeMODBUS( String hexString)
uwp   async Task<int> writeMODBUS( string hexString)
py    writeMODBUS( hexString)
php   function writeMODBUS( $hexString)
es    async writeMODBUS( hexString)
dnp   int writeMODBUS( string hexString)
cp    int writeMODBUS( string hexString)
cmd   YSerialPort target writeMODBUS hexString
```

The message must start with the slave address. The MODBUS CRC/LRC is automatically added by the function. This function does not wait for a reply.

Parameters :

hexString a hexadecimal message string, including device address but no CRC/LRC

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

serialport→writeStr() YSerialPort

Sends an ASCII string to the serial port, as is.

```
js    function writeStr( text)
cpp   int writeStr( string text)
m     -(int) writeStr : (NSString*) text
pas   LongInt writeStr( text: string): LongInt
vb    function writeStr( ByVal text As String) As Integer
cs    int writeStr( string text)
java  int writeStr( String text)
uwp   async Task<int> writeStr( string text)
py    writeStr( text)
php   function writeStr( $text)
es    async writeStr( text)
dnp   int writeStr( string text)
cp    int writeStr( string text)
cmd   YSerialPort target writeStr text
```

Parameters :

text the text string to send

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

24.4. Class YFiles

Filesystem control interface, available for instance in the Yocto-Buzzer, the Yocto-Color-V2, the Yocto-RS485-V2 or the YoctoHub-Ethernet

The YFiles class is used to access the filesystem embedded on some Yoctopuce devices. This filesystem makes it possible for instance to design a custom web UI (for networked devices) or to add fonts (on display devices).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_files.js'></script>
cpp	#include "yocto_files.h"
m	#import "yocto_files.h"
pas	uses yocto_files;
vb	yocto_files.vb
cs	yocto_files.cs
java	import com.yoctopuce.YoctoAPI.YFiles;
uwp	import com.yoctopuce.YoctoAPI.YFiles;
py	from yocto_files import *
php	require_once('yocto_files.php');
es	in HTML: <script src="../../lib/yocto_files.js"></script> in node.js: require('yoctolib-es2017/yocto_files.js');
dnp	import YoctoProxyAPI.YFilesProxy
cp	#include "yocto_files_proxy.h"
vi	YFiles.vi
ml	import YoctoProxyAPI.YFilesProxy

Global functions
YFiles.FindFiles(**func**) Retrieves a filesystem for a given identifier.
YFiles.FindFilesInContext(**yctx**, **func**) Retrieves a filesystem for a given identifier in a YAPI context.
YFiles.FirstFiles() Starts the enumeration of filesystems currently accessible.
YFiles.FirstFilesInContext(**yctx**) Starts the enumeration of filesystems currently accessible.
YFiles.GetSimilarFunctions() Enumerates all functions of type Files available on the devices currently reachable by the library, and returns their unique hardware ID.

`YFiles` properties
files→AdvertisedValue *[read-only]* Short string representing the current state of the function.
files→FilesCount *[read-only]* Number of files currently loaded in the filesystem.
files→FriendlyName *[read-only]* Global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.
files→FunctionId *[read-only]* Hardware identifier of the filesystem, without reference to the module.

files→HardwareId *[read-only]*

Unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`.

files→IsOnline *[read-only]*

Checks if the function is currently reachable.

files→LogicalName *[writable]*

Logical name of the function.

files→SerialNumber *[read-only]*

Serial number of the module, as set by the factory.

`YFiles` methods

files→clearCache()

Invalidates the cache.

files→describe()

Returns a short text that describes unambiguously the instance of the filesystem in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

files→download(pathname)

Downloads the requested file and returns a binary buffer with its content.

files→download_async(pathname, callback, context)

Downloads the requested file and returns a binary buffer with its content.

files→fileExist(filename)

Test if a file exist on the filesystem of the module.

files→format_fs()

Reinitialize the filesystem to its clean, unfragmented, empty state.

files→get_advertisedValue()

Returns the current value of the filesystem (no more than 6 characters).

files→get_errorMessage()

Returns the error message of the latest error with the filesystem.

files→get_errorType()

Returns the numerical error code of the latest error with the filesystem.

files→get_filesCount()

Returns the number of files currently loaded in the filesystem.

files→get_freeSpace()

Returns the free space for uploading new files to the filesystem, in bytes.

files→get_friendlyName()

Returns a global identifier of the filesystem in the format `MODULE_NAME.FUNCTION_NAME`.

files→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

files→get_functionId()

Returns the hardware identifier of the filesystem, without reference to the module.

files→get_hardwareId()

Returns the unique hardware identifier of the filesystem in the form `SERIAL.FUNCTIONID`.

files→get_list(pattern)

Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.

files→get_logicalName()

Returns the logical name of the filesystem.

files→get_module()

Gets the `YModule` object for the device on which the function is located.

files→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

files→get_serialNumber()

Returns the serial number of the module, as set by the factory.

files→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

files→isOnline()

Checks if the filesystem is currently reachable, without raising any error.

files→isOnline_async(callback, context)

Checks if the filesystem is currently reachable, without raising any error (asynchronous version).

files→isReadOnly()

Test if the function is readOnly.

files→load(msValidity)

Preloads the filesystem cache with a specified validity duration.

files→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

files→load_async(msValidity, callback, context)

Preloads the filesystem cache with a specified validity duration (asynchronous version).

files→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

files→nextFiles()

Continues the enumeration of filesystems started using `yFirstFiles()`.

files→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

files→remove(pathname)

Deletes a file, given by its full path name, from the filesystem.

files→set_logicalName(newval)

Changes the logical name of the filesystem.

files→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

files→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

files→upload(pathname, content)

Uploads a file to the filesystem, to the specified full path name.

files→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YFiles.FindFiles() YFiles

YFiles.FindFiles()

Retrieves a filesystem for a given identifier.

```
js    function yFindFiles( func)
cpp   YFiles* yFindFiles( string func)
m     +(YFiles*) FindFiles : (NSString*) func
pas   TYFiles yFindFiles( func: string): TYFiles
vb    function yFindFiles( ByVal func As String) As YFiles
cs    static YFiles FindFiles( string func)
java  static YFiles FindFiles( String func)
uwp   static YFiles FindFiles( string func)
py    FindFiles( func)
php   function yFindFiles( $func)
es    static FindFiles( func)
dnp   static YFilesProxy FindFiles( string func)
cp    static YFilesProxy * FindFiles( string 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 filesystem is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YFiles.isOnline()` to test if the filesystem is indeed online at a given time. In case of ambiguity when looking for a filesystem by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the filesystem, for instance `YBUZZER2.files`.

Returns :

a `YFiles` object allowing you to drive the filesystem.

YFiles.FindFilesInContext() YFiles

YFiles.FindFilesInContext()

Retrieves a filesystem for a given identifier in a YAPI context.

```
java  static YFiles FindFilesInContext( YAPIContext yctx, String func)
uwp   static YFiles FindFilesInContext( YAPIContext yctx, string func)
es    static FindFilesInContext( yctx, func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the filesystem is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YFiles.isOnline()` to test if the filesystem is indeed online at a given time. In case of ambiguity when looking for a filesystem by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the filesystem, for instance `YBUZZER2.files`.

Returns :

a `YFiles` object allowing you to drive the filesystem.

YFiles.FirstFiles() YFiles

YFiles.FirstFiles()

Starts the enumeration of filesystems currently accessible.

js	function **yFirstFiles**( )
cpp	YFiles * **yFirstFiles**( )
m	+(YFiles*) **FirstFiles**
pas	TYFiles **yFirstFiles**( ): TYFiles
vb	function **yFirstFiles**( ) As YFiles
cs	static YFiles **FirstFiles**( )
java	static YFiles **FirstFiles**( )
uwp	static YFiles **FirstFiles**( )
py	**FirstFiles**( )
php	function **yFirstFiles**( )
es	static **FirstFiles**( )

Use the method `YFiles.nextFiles()` to iterate on next filesystems.

Returns :

a pointer to a `YFiles` object, corresponding to the first filesystem currently online, or a `null` pointer if there are none.

YFiles.FirstFilesInContext() YFiles.FirstFilesInContext() — YFiles

Starts the enumeration of filesystems currently accessible.

```
java  static YFiles FirstFilesInContext( YAPIContext yctx)
uwp   static YFiles FirstFilesInContext( YAPIContext yctx)
es    static FirstFilesInContext( yctx)
```

Use the method `YFiles.nextFiles()` to iterate on next filesystems.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YFiles` object, corresponding to the first filesystem currently online, or a `null` pointer if there are none.

YFiles.GetSimilarFunctions() YFiles

YFiles.GetSimilarFunctions()

Enumerates all functions of type Files available on the devices currently reachable by the library, and returns their unique hardware ID.

dnp static new string[] **GetSimilarFunctions**( )

cp static vector<string> **GetSimilarFunctions**( )

Each of these IDs can be provided as argument to the method `YFiles.FindFiles` to obtain an object that can control the corresponding device.

Returns :

an array of strings, each string containing the unique hardwareId of a device function currently connected.

files→AdvertisedValue YFiles

Short string representing the current state of the function.

dnp string **AdvertisedValue**

files→FilesCount YFiles

Number of files currently loaded in the filesystem.

dnp int **FilesCount**

files→FriendlyName YFiles

Global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

dnp string **FriendlyName**

The returned string uses the logical names of the module and of the function if they are defined, otherwise the serial number of the module and the hardware identifier of the function (for example: `MyCustomName.relay1`)

files→FunctionId YFiles

Hardware identifier of the filesystem, without reference to the module.

dnp string **FunctionId**

For example `relay1`

files→HardwareId YFiles

Unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`.

dnp string **HardwareId**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example `RELAYLO1-123456.relay1`).

files→IsOnline YFiles

Checks if the function is currently reachable.

dnp bool **IsOnline**

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the function.

files→LogicalName YFiles

Logical name of the function.

dnp string **LogicalName**

Writable. You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

files→SerialNumber YFiles

Serial number of the module, as set by the factory.

dnp string **SerialNumber**

files→clearCache() YFiles

Invalidates the cache.

js	function **clearCache**( )
cpp	void **clearCache**( )
m	-(void) **clearCache**
pas	**clearCache**( )
vb	procedure **clearCache**( )
cs	void **clearCache**( )
java	void **clearCache**( )
py	**clearCache**( )
php	function **clearCache**( )
es	async **clearCache**( )

Invalidates the cache of the filesystem attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

files→describe() YFiles

Returns a short text that describes unambiguously the instance of the filesystem in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

js	function **describe**( )
cpp	string **describe**( )
m	-(NSString*) **describe**
pas	string **describe**( ): string
vb	function **describe**( ) As String
cs	string **describe**( )
java	String **describe**( )
py	**describe**( )
php	function **describe**( )
es	async **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :
a string that describes the filesystem (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

files→download() YFiles

Downloads the requested file and returns a binary buffer with its content.

```
js    function download( pathname)
cpp   string download( string pathname)
m     -(NSMutableData*) download : (NSString*) pathname
pas   TByteArray download( pathname: string): TByteArray
vb    function download( ByVal pathname As String) As Byte
cs    byte[] download( string pathname)
java  byte[] download( String pathname)
uwp   async Task<byte[]> download( string pathname)
py    download( pathname)
php   function download( $pathname)
es    async download( pathname)
dnp   byte[] download( string pathname)
cp    string download( string pathname)
cmd   YFiles target download pathname
```

Parameters :

pathname path and name of the file to download

Returns :

a binary buffer with the file content

On failure, throws an exception or returns an empty content.

files→download_async() YFiles

Downloads the requested file and returns a binary buffer with its content.

```
js function download_async( pathname, callback, context)
```

This is the asynchronous version that uses a callback to pass the result when the download is completed.

Parameters :

pathname path and name of the new file to load

callback callback function that is invoked when the w The callback function receives three arguments: - the user-specific context object - the YFiles object whose download_async was invoked - a binary buffer with the file content

context user-specific object that is passed as-is to the callback function

Returns :

nothing.

files→fileExist() YFiles

Test if a file exist on the filesystem of the module.

```
js    function fileExist( filename)
cpp   bool fileExist( string filename)
m     -(bool) fileExist : (NSString*) filename
pas   boolean fileExist( filename: string): boolean
vb    function fileExist( ByVal filename As String) As Boolean
cs    bool fileExist( string filename)
java  boolean fileExist( String filename)
uwp   async Task<bool> fileExist( string filename)
py    fileExist( filename)
php   function fileExist( $filename)
es    async fileExist( filename)
dnp   bool fileExist( string filename)
cp    bool fileExist( string filename)
cmd   YFiles target fileExist filename
```

Parameters :

filename the file name to test.

Returns :

a true if the file exist, false otherwise.

On failure, throws an exception.

files→format_fs() YFiles

Reinitialize the filesystem to its clean, unfragmented, empty state.

js	function **format_fs**( )
cpp	int **format_fs**( )
m	-(int) **format_fs**
pas	LongInt **format_fs**( ): LongInt
vb	function **format_fs**( ) As Integer
cs	int **format_fs**( )
java	int **format_fs**( )
uwp	async Task<int> **format_fs**( )
py	**format_fs**( )
php	function **format_fs**( )
es	async **format_fs**( )
dnp	int **format_fs**( )
cp	int **format_fs**( )
cmd	YFiles **target format_fs**

All files previously uploaded are permanently lost.

Returns :
`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→get_advertisedValue() YFiles
files→advertisedValue()

Returns the current value of the filesystem (no more than 6 characters).

```
js    function get_advertisedValue( )
cpp   string get_advertisedValue( )
m     -(NSString*) advertisedValue
pas   string get_advertisedValue( ): string
vb    function get_advertisedValue( ) As String
cs    string get_advertisedValue( )
java  String get_advertisedValue( )
uwp   async Task<string> get_advertisedValue( )
py    get_advertisedValue( )
php   function get_advertisedValue( )
es    async get_advertisedValue( )
dnp   string get_advertisedValue( )
cp    string get_advertisedValue( )
cmd   YFiles target get_advertisedValue
```

Returns :
a string corresponding to the current value of the filesystem (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

files→get_errorMessage() YFiles
files→errorMessage()

Returns the error message of the latest error with the filesystem.

```
js    function get_errorMessage( )
cpp   string get_errorMessage( )
m     -(NSString*) errorMessage
pas   string get_errorMessage( ): string
vb    function get_errorMessage( ) As String
cs    string get_errorMessage( )
java  String get_errorMessage( )
py    get_errorMessage( )
php   function get_errorMessage( )
es    get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the filesystem object

files→get_errorType() YFiles
files→errorType()

Returns the numerical error code of the latest error with the filesystem.

js	function **get_errorType**( )
cpp	YRETCODE **get_errorType**( )
m	-(YRETCODE) errorType
pas	YRETCODE **get_errorType**( ): YRETCODE
vb	function **get_errorType**( ) As YRETCODE
cs	YRETCODE **get_errorType**( )
java	int **get_errorType**( )
py	**get_errorType**( )
php	function **get_errorType**( )
es	**get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the filesystem object

files→get_filesCount() YFiles
files→filesCount()

Returns the number of files currently loaded in the filesystem.

```
js    function get_filesCount( )
cpp   int get_filesCount( )
m     -(int) filesCount
pas   LongInt get_filesCount( ): LongInt
vb    function get_filesCount( ) As Integer
cs    int get_filesCount( )
java  int get_filesCount( )
uwp   async Task<int> get_filesCount( )
py    get_filesCount( )
php   function get_filesCount( )
es    async get_filesCount( )
dnp   int get_filesCount( )
cp    int get_filesCount( )
cmd   YFiles target get_filesCount
```

Returns :
an integer corresponding to the number of files currently loaded in the filesystem

On failure, throws an exception or returns `Y_FILESCOUNT_INVALID`.

files→get_freeSpace() YFiles
files→freeSpace()

Returns the free space for uploading new files to the filesystem, in bytes.

js	function **get_freeSpace**( )
cpp	int **get_freeSpace**( )
m	-(int) freeSpace
pas	LongInt **get_freeSpace**( ): LongInt
vb	function **get_freeSpace**( ) As Integer
cs	int **get_freeSpace**( )
java	int **get_freeSpace**( )
uwp	async Task<int> **get_freeSpace**( )
py	**get_freeSpace**( )
php	function **get_freeSpace**( )
es	async **get_freeSpace**( )
dnp	int **get_freeSpace**( )
cp	int **get_freeSpace**( )
cmd	YFiles **target get_freeSpace**

Returns :
an integer corresponding to the free space for uploading new files to the filesystem, in bytes

On failure, throws an exception or returns `Y_FREESPACE_INVALID`.

files→get_friendlyName() YFiles
files→friendlyName()

Returns a global identifier of the filesystem in the format `MODULE_NAME.FUNCTION_NAME`.

js	function **get_friendlyName**( )
cpp	string **get_friendlyName**( )
m	-(NSString*) friendlyName
cs	string **get_friendlyName**( )
java	String **get_friendlyName**( )
py	**get_friendlyName**( )
php	function **get_friendlyName**( )
es	async **get_friendlyName**( )
dnp	string **get_friendlyName**( )
cp	string **get_friendlyName**( )

The returned string uses the logical names of the module and of the filesystem if they are defined, otherwise the serial number of the module and the hardware identifier of the filesystem (for example: `MyCustomName.relay1`)

Returns :
a string that uniquely identifies the filesystem using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

files→get_functionDescriptor() YFiles
files→functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

js	function **get_functionDescriptor**( )
cpp	YFUN_DESCR **get_functionDescriptor**( )
m	-(YFUN_DESCR) functionDescriptor
pas	YFUN_DESCR **get_functionDescriptor**( ): YFUN_DESCR
vb	function **get_functionDescriptor**( ) As YFUN_DESCR
cs	YFUN_DESCR **get_functionDescriptor**( )
java	String **get_functionDescriptor**( )
py	**get_functionDescriptor**( )
php	function **get_functionDescriptor**( )
es	async **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :
an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

files→get_functionId() YFiles
files→functionId()

Returns the hardware identifier of the filesystem, without reference to the module.

js	function **get_functionId**( )
cpp	string **get_functionId**( )
m	-(NSString*) functionId
vb	function **get_functionId**( ) As String
cs	string **get_functionId**( )
java	String **get_functionId**( )
py	**get_functionId**( )
php	function **get_functionId**( )
es	async **get_functionId**( )
dnp	string **get_functionId**( )
cp	string **get_functionId**( )

For example `relay1`

Returns :
a string that identifies the filesystem (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

files→get_hardwareId() YFiles
files→hardwareId()

Returns the unique hardware identifier of the filesystem in the form `SERIAL.FUNCTIONID`.

js	function **get_hardwareId**( )
cpp	string **get_hardwareId**( )
m	-(NSString*) hardwareId
vb	function **get_hardwareId**( ) As String
cs	string **get_hardwareId**( )
java	String **get_hardwareId**( )
py	**get_hardwareId**( )
php	function **get_hardwareId**( )
es	async **get_hardwareId**( )
dnp	string **get_hardwareId**( )
cp	string **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the filesystem (for example `RELAYLO1-123456.relay1`).

Returns :
a string that uniquely identifies the filesystem (ex: `RELAYLO1-123456.relay1`)

On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

files→get_list() YFiles
files→list()

Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.

```
js    function get_list( pattern)
cpp   vector<YFileRecord> get_list( string pattern)
m     -(NSMutableArray*) list : (NSString*) pattern
pas   TYFileRecordArray get_list( pattern: string): TYFileRecordArray
vb    function get_list( ByVal pattern As String) As List
cs    List<YFileRecord> get_list( string pattern)
java  ArrayList<YFileRecord> get_list( String pattern)
uwp   async Task<List<YFileRecord>> get_list( string pattern)
py    get_list( pattern)
php   function get_list( $pattern)
es    async get_list( pattern)
dnp   YFileRecordProxy[] get_list( string pattern)
cp    vector<YFileRecordProxy> get_list( string pattern)
cmd   YFiles target get_list pattern
```

Parameters :

pattern an optional filter pattern, using star and question marks as wild cards. When an empty pattern is provided, all file records are returned.

Returns :

a list of `YFileRecord` objects, containing the file path and name, byte size and 32-bit CRC of the file content.

On failure, throws an exception or returns an empty list.

files→get_logicalName() YFiles
files→logicalName()

Returns the logical name of the filesystem.

js	function **get_logicalName**( )
cpp	string **get_logicalName**( )
m	-(NSString*) logicalName
pas	string **get_logicalName**( ): string
vb	function **get_logicalName**( ) As String
cs	string **get_logicalName**( )
java	String **get_logicalName**( )
uwp	async Task<string> **get_logicalName**( )
py	**get_logicalName**( )
php	function **get_logicalName**( )
es	async **get_logicalName**( )
dnp	string **get_logicalName**( )
cp	string **get_logicalName**( )
cmd	YFiles **target get_logicalName**

Returns :
a string corresponding to the logical name of the filesystem.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

files→get_module() YFiles
files→module()

Gets the `YModule` object for the device on which the function is located.

js	function **get_module**( )
cpp	YModule * **get_module**( )
m	-(YModule*) module
pas	TYModule **get_module**( ): TYModule
vb	function **get_module**( ) As YModule
cs	YModule **get_module**( )
java	YModule **get_module**( )
py	**get_module**( )
php	function **get_module**( )
es	async **get_module**( )
dnp	YModuleProxy **get_module**( )
cp	YModuleProxy * **get_module**( )

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

Returns :
an instance of `YModule`

files→get_module_async() YFiles
files→module_async()

Gets the `YModule` object for the device on which the function is located (asynchronous version).

```
js  function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as online.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking Firefox JavaScript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous JavasSript calls for more details.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

files→get_serialNumber() YFiles
files→serialNumber()

Returns the serial number of the module, as set by the factory.

```
js      function get_serialNumber( )
cpp     string get_serialNumber( )
m       -(NSString*) serialNumber
pas     string get_serialNumber( ): string
vb      function get_serialNumber( ) As String
cs      string get_serialNumber( )
java    String get_serialNumber( )
uwp     async Task<string> get_serialNumber( )
py      get_serialNumber( )
php     function get_serialNumber( )
es      async get_serialNumber( )
dnp     string get_serialNumber( )
cp      string get_serialNumber( )
cmd     YFiles target get_serialNumber
```

Returns :
a string corresponding to the serial number of the module, as set by the factory.

On failure, throws an exception or returns YModule.SERIALNUMBER_INVALID.

files→get_userData() YFiles
files→userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

js	function **get_userData**( )
cpp	void * **get_userData**( )
m	-(id) userData
pas	Tobject **get_userData**( ): Tobject
vb	function **get_userData**( ) As Object
cs	object **get_userData**( )
java	Object **get_userData**( )
py	**get_userData**( )
php	function **get_userData**( )
es	async **get_userData**( )

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

files→isOnline() YFiles

Checks if the filesystem is currently reachable, without raising any error.

js	function **isOnline**( )
cpp	bool **isOnline**( )
m	-(BOOL) **isOnline**
pas	boolean **isOnline**( ): boolean
vb	function **isOnline**( ) As Boolean
cs	bool **isOnline**( )
java	boolean **isOnline**( )
py	**isOnline**( )
php	function **isOnline**( )
es	async **isOnline**( )
dnp	bool **isOnline**( )
cp	bool **isOnline**( )

If there is a cached value for the filesystem 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 filesystem.

Returns :
`true` if the filesystem can be reached, and `false` otherwise

files→isOnline_async() YFiles

Checks if the filesystem is currently reachable, without raising any error (asynchronous version).

```
js function isOnline_async( callback, context)
```

If there is a cached value for the filesystem 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.

files→isReadOnly() YFiles

Test if the function is readOnly.

cpp	bool **isReadOnly**( )
m	-(bool) **isReadOnly**
pas	boolean **isReadOnly**( ): boolean
vb	function **isReadOnly**( ) As Boolean
cs	bool **isReadOnly**( )
java	boolean **isReadOnly**( )
uwp	async Task<bool> **isReadOnly**( )
py	**isReadOnly**( )
php	function **isReadOnly**( )
es	async **isReadOnly**( )
dnp	bool **isReadOnly**( )
cp	bool **isReadOnly**( )
cmd	YFiles **target isReadOnly**

Return `true` if the function is write protected or that the function is not available.

Returns :
`true` if the function is readOnly or not online.

files→load() YFiles

Preloads the filesystem cache with a specified validity duration.

```
js    function load( msValidity)
cpp   YRETCODE load( int msValidity)
m     -(YRETCODE) load : (u64) msValidity
pas   YRETCODE load( msValidity: u64): YRETCODE
vb    function load( ByVal msValidity As Long) As YRETCODE
cs    YRETCODE load( ulong msValidity)
java  int load( long msValidity)
py    load( msValidity)
php   function load( $msValidity)
es    async load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

files→loadAttribute() YFiles

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
js    function loadAttribute( attrName)
cpp   string loadAttribute( string attrName)
m     -(NSString*) loadAttribute : (NSString*) attrName
pas   string loadAttribute( attrName: string): string
vb    function loadAttribute( ByVal attrName As String) As String
cs    string loadAttribute( string attrName)
java  String loadAttribute( String attrName)
uwp   async Task<string> loadAttribute( string attrName)
py    loadAttribute( attrName)
php   function loadAttribute( $attrName)
es    async loadAttribute( attrName)
dnp   string loadAttribute( string attrName)
cp    string loadAttribute( string attrName)
```

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

files→load_async() YFiles

Preloads the filesystem cache with a specified validity duration (asynchronous version).

```
js function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking the JavaScript virtual machine.

Parameters :

msValidity an integer corresponding to the validity of the loaded function parameters, in milliseconds

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

files→muteValueCallbacks() YFiles

Disables the propagation of every new advertised value to the parent hub.

js	function **muteValueCallbacks**( )
cpp	int **muteValueCallbacks**( )
m	-(int) **muteValueCallbacks**
pas	LongInt **muteValueCallbacks**( ): LongInt
vb	function **muteValueCallbacks**( ) As Integer
cs	int **muteValueCallbacks**( )
java	int **muteValueCallbacks**( )
uwp	async Task<int> **muteValueCallbacks**( )
py	**muteValueCallbacks**( )
php	function **muteValueCallbacks**( )
es	async **muteValueCallbacks**( )
dnp	int **muteValueCallbacks**( )
cp	int **muteValueCallbacks**( )
cmd	YFiles **target muteValueCallbacks**

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

files→nextFiles() YFiles

Continues the enumeration of filesystems started using `yFirstFiles()`.

js	function **nextFiles**( )
cpp	YFiles * **nextFiles**( )
m	-(YFiles*) **nextFiles**
pas	TYFiles **nextFiles**( ): TYFiles
vb	function **nextFiles**( ) As YFiles
cs	YFiles **nextFiles**( )
java	YFiles **nextFiles**( )
uwp	YFiles **nextFiles**( )
py	**nextFiles**( )
php	function **nextFiles**( )
es	**nextFiles**( )

Caution: You can't make any assumption about the returned filesystems order. If you want to find a specific a filesystem, use `Files.findFiles()` and a hardwareID or a logical name.

Returns :

a pointer to a `YFiles` object, corresponding to a filesystem currently online, or a `null` pointer if there are no more filesystems to enumerate.

files→registerValueCallback() YFiles

Registers the callback function that is invoked on every change of advertised value.

js	function **registerValueCallback**(**callback**)
cpp	int **registerValueCallback**(YFilesValueCallback **callback**)
m	-(int) **registerValueCallback** : (YFilesValueCallback) **callback**
pas	LongInt **registerValueCallback**(**callback**: TYFilesValueCallback): LongInt
vb	function **registerValueCallback**(ByVal **callback** As YFilesValueCallback) As Integer
cs	int **registerValueCallback**(ValueCallback **callback**)
java	int **registerValueCallback**(UpdateCallback **callback**)
uwp	async Task<int> **registerValueCallback**(ValueCallback **callback**)
py	**registerValueCallback**(**callback**)
php	function **registerValueCallback**(**$callback**)
es	async **registerValueCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

files→remove() YFiles

Deletes a file, given by its full path name, from the filesystem.

```
js    function remove( pathname)
cpp   int remove( string pathname)
m     -(int) remove : (NSString*) pathname
pas   LongInt remove( pathname: string): LongInt
vb    function remove( ByVal pathname As String) As Integer
cs    int remove( string pathname)
java  int remove( String pathname)
uwp   async Task<int> remove( string pathname)
py    remove( pathname)
php   function remove( $pathname)
es    async remove( pathname)
dnp   int remove( string pathname)
cp    int remove( string pathname)
cmd   YFiles target remove pathname
```

Because of filesystem fragmentation, deleting a file may not always free up the whole space used by the file. However, rewriting a file with the same path name will always reuse any space not freed previously. If you need to ensure that no space is taken by previously deleted files, you can use `format_fs` to fully reinitialize the filesystem.

Parameters :

pathname path and name of the file to remove.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→set_logicalName() YFiles
files→setLogicalName()

Changes the logical name of the filesystem.

```
js    function set_logicalName( newval)
cpp   int set_logicalName( string newval)
m     -(int) setLogicalName : (NSString*) newval
pas   integer set_logicalName( newval: string): integer
vb    function set_logicalName( ByVal newval As String) As Integer
cs    int set_logicalName( string newval)
java  int set_logicalName( String newval)
uwp   async Task<int> set_logicalName( string newval)
py    set_logicalName( newval)
php   function set_logicalName( $newval)
es    async set_logicalName( newval)
dnp   int set_logicalName( string newval)
cp    int set_logicalName( string newval)
cmd   YFiles target set_logicalName newval
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the filesystem.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→set_userData() YFiles
files→setUserData()

Stores a user context provided as argument in the userData attribute of the function.

```
js    function set_userData( data)
cpp   void set_userData( void * data)
m     -(void) setUserData : (id) data
pas   set_userData( data: Tobject)
vb    procedure set_userData( ByVal data As Object)
cs    void set_userData( object data)
java  void set_userData( Object data)
py    set_userData( data)
php   function set_userData( $data)
es    async set_userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

files→unmuteValueCallbacks() YFiles

Re-enables the propagation of every new advertised value to the parent hub.

js	function **unmuteValueCallbacks**( )
cpp	int **unmuteValueCallbacks**( )
m	-(int) **unmuteValueCallbacks**
pas	LongInt **unmuteValueCallbacks**( ): LongInt
vb	function **unmuteValueCallbacks**( ) As Integer
cs	int **unmuteValueCallbacks**( )
java	int **unmuteValueCallbacks**( )
uwp	async Task<int> **unmuteValueCallbacks**( )
py	**unmuteValueCallbacks**( )
php	function **unmuteValueCallbacks**( )
es	async **unmuteValueCallbacks**( )
dnp	int **unmuteValueCallbacks**( )
cp	int **unmuteValueCallbacks**( )
cmd	YFiles **target unmuteValueCallbacks**

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

files→upload() YFiles

Uploads a file to the filesystem, to the specified full path name.

```
js    function upload( pathname, content)
cpp   int upload( string pathname, string content)
m     -(int) upload : (NSString*) pathname
                    : (NSData*) content
pas   LongInt upload( pathname: string, content: TByteArray): LongInt
vb    procedure upload( ByVal pathname As String, ByVal content As Byte()
cs    int upload( string pathname, byte[] content)
java  int upload( String pathname, byte[] content)
uwp   async Task<int> upload( string pathname, byte[] content)
py    upload( pathname, content)
php   function upload( $pathname, $content)
es    async upload( pathname, content)
dnp   int upload( string pathname, byte[] content)
cp    int upload( string pathname, string content)
cmd   YFiles target upload pathname content
```

If a file already exists with the same path name, its content is overwritten.

Parameters :

pathname path and name of the new file to create

content binary buffer with the content to set

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→wait_async() YFiles

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js  function wait_async( callback, context)
es  wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

24.5. Class YGenericSensor

GenericSensor control interface, available for instance in the Yocto-0-10V-Rx, the Yocto-4-20mA-Rx, the Yocto-RS485-V2 or the Yocto-milliVolt-Rx

The `YGenericSensor` class allows you to read and configure Yoctopuce signal transducers. It inherits from `YSensor` class the core functions to read measurements, to register callback functions, to access the autonomous datalogger. This class adds the ability to configure the automatic conversion between the measured signal and the corresponding engineering unit.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_genericsensor.js'></script>
cpp	#include "yocto_genericsensor.h"
m	#import "yocto_genericsensor.h"
pas	uses yocto_genericsensor;
vb	yocto_genericsensor.vb
cs	yocto_genericsensor.cs
java	import com.yoctopuce.YoctoAPI.YGenericSensor;
uwp	import com.yoctopuce.YoctoAPI.YGenericSensor;
py	from yocto_genericsensor import *
php	require_once('yocto_genericsensor.php');
es	in HTML: <script src="../../lib/yocto_genericsensor.js"></script> in node.js: require('yoctolib-es2017/yocto_genericsensor.js');
dnp	import YoctoProxyAPI.YGenericSensorProxy
cp	#include "yocto_genericsensor_proxy.h"
vi	YGenericSensor.vi
ml	import YoctoProxyAPI.YGenericSensorProxy

Global functions

YGenericSensor.FindGenericSensor(**func**)

Retrieves a generic sensor for a given identifier.

YGenericSensor.FindGenericSensorInContext(**yctx**, **func**)

Retrieves a generic sensor for a given identifier in a YAPI context.

YGenericSensor.FirstGenericSensor()

Starts the enumeration of generic sensors currently accessible.

YGenericSensor.FirstGenericSensorInContext(**yctx**)

Starts the enumeration of generic sensors currently accessible.

YGenericSensor.GetSimilarFunctions()

Enumerates all functions of type GenericSensor available on the devices currently reachable by the library, and returns their unique hardware ID.

`YGenericSensor` properties

genericsensor→AdvMode *[writable]*

Measuring mode used for the advertised value pushed to the parent hub.

genericsensor→AdvertisedValue *[read-only]*

Short string representing the current state of the function.

genericsensor→Enabled *[writable]*

Activation state of this input.

genericsensor→FriendlyName *[read-only]*

Global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

genericsensor→FunctionId *[read-only]*

Hardware identifier of the sensor, without reference to the module.

genericsensor→HardwareId *[read-only]*

Unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`.

genericsensor→IsOnline *[read-only]*

Checks if the function is currently reachable.

genericsensor→LogFrequency *[writable]*

Datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

genericsensor→LogicalName *[writable]*

Logical name of the function.

genericsensor→ReportFrequency *[writable]*

Timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

genericsensor→Resolution *[writable]*

Resolution of the measured values.

genericsensor→SerialNumber *[read-only]*

Serial number of the module, as set by the factory.

genericsensor→SignalBias *[writable]*

Electric signal bias for zero shift adjustment.

genericsensor→SignalRange *[writable]*

Input signal range used by the sensor.

genericsensor→SignalSampling *[writable]*

Electric signal sampling method to use.

genericsensor→SignalUnit *[read-only]*

Measuring unit of the electrical signal used by the sensor.

genericsensor→ValueRange *[writable]*

Physical value range measured by the sensor.

`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→clearCache()

Invalidates the cache.

genericsensor→describe()

Returns a short text that describes unambiguously the instance of the generic sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

genericsensor→get_advMode()

Returns the measuring mode used for the advertised value pushed to the parent hub.

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_dataLogger()

Returns the `YDatalogger` object of the device hosting the sensor.

genericsensor→get_enabled()

Returns the activation state of this input.

genericsensor→get_errorMessage()

Returns the error message of the latest error with the generic sensor.

genericsensor→get_errorType()

Returns the numerical error code of the latest error with the generic sensor.

genericsensor→get_friendlyName()

Returns a global identifier of the generic sensor in the format `MODULE_NAME.FUNCTION_NAME`.

genericsensor→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

genericsensor→get_functionId()

Returns the hardware identifier of the generic sensor, without reference to the module.

genericsensor→get_hardwareId()

Returns the unique hardware identifier of the generic sensor in the form `SERIAL.FUNCTIONID`.

genericsensor→get_highestValue()

Returns the maximal value observed for the measure since the device was started.

genericsensor→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

genericsensor→get_logicalName()

Returns the logical name of the generic sensor.

genericsensor→get_lowestValue()

Returns the minimal value observed for the measure since the device was started.

genericsensor→get_module()

Gets the `YModule` object for the device on which the function is located.

genericsensor→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

genericsensor→get_recordedData(startTime, endTime)

Retrieves a `YDataSet` 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_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

genericsensor→get_serialNumber()

Returns the serial number of the module, as set by the factory.

genericsensor→get_signalBias()

Returns the electric signal bias for zero shift adjustment.

genericsensor→get_signalRange()

Returns the input signal range used by the sensor.

genericsensor→get_signalSampling()

Returns the electric signal sampling method to use.

genericsensor→get_signalUnit()

Returns the measuring unit of the electrical signal used by the sensor.

genericsensor→get_signalValue()

Returns the current value of the electrical signal measured by the sensor.

genericsensor→get_unit()

Returns the measuring unit for the measure.

genericsensor→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

genericsensor→get_valueRange()

Returns the physical value range measured by the sensor.

genericsensor→isOnline()

Checks if the generic sensor is currently reachable, without raising any error.

genericsensor→isOnline_async(callback, context)

Checks if the generic sensor is currently reachable, without raising any error (asynchronous version).

genericsensor→isReadOnly()

Test if the function is readOnly.

genericsensor→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

genericsensor→load(msValidity)

Preloads the generic sensor cache with a specified validity duration.

genericsensor→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

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→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

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_advMode(newval)

Changes the measuring mode used for the advertised value pushed to the parent hub.

genericsensor→set_enabled(newval)

Changes the activation state of this input.

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(resolution, newval)

Modifies the resolution of the measured physical values.

genericsensor→set_signalBias(newval)

Changes the electric signal bias for zero shift adjustment.

genericsensor→set_signalRange(newval)

Changes the input signal range used by the sensor.

genericsensor→set_signalSampling(newval)

Changes the electric signal sampling method to use.

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 output value range, corresponding to the physical value measured by the sensor.

genericsensor→startDataLogger()

Starts the data logger on the device.

genericsensor→stopDataLogger()

Stops the datalogger on the device.

genericsensor→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

genericsensor→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

genericsensor→zeroAdjust()

Adjusts the signal bias so that the current signal value is need precisely as zero.

YGenericSensor.FindGenericSensor() YGenericSensor

YGenericSensor.FindGenericSensor()

Retrieves a generic sensor for a given identifier.

```
js    function yFindGenericSensor( func)
cpp   YGenericSensor* yFindGenericSensor( string func)
m     +(YGenericSensor*) FindGenericSensor : (NSString*) func
pas   TYGenericSensor yFindGenericSensor( func: string): TYGenericSensor
vb    function yFindGenericSensor( ByVal func As String) As YGenericSensor
cs    static YGenericSensor FindGenericSensor( string func)
java  static YGenericSensor FindGenericSensor( String func)
uwp   static YGenericSensor FindGenericSensor( string func)
py    FindGenericSensor( func)
php   function yFindGenericSensor( $func)
es    static FindGenericSensor( func)
dnp   static YGenericSensorProxy FindGenericSensor( string func)
cp    static YGenericSensorProxy * FindGenericSensor( string 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.

If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the generic sensor, for instance `RX010V01.genericSensor1`.

Returns :

a `YGenericSensor` object allowing you to drive the generic sensor.

YGenericSensor.FindGenericSensorInContext() YGenericSensor

YGenericSensor.FindGenericSensorInContext()

Retrieves a generic sensor for a given identifier in a YAPI context.

```
java  static YGenericSensor FindGenericSensorInContext( YAPIContext yctx,
                                                        String func)
uwp   static YGenericSensor FindGenericSensorInContext( YAPIContext yctx,
                                                        string func)
es    static FindGenericSensorInContext( yctx, func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the 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 :

yctx a YAPI context

func a string that uniquely characterizes the generic sensor, for instance `RX010V01.genericSensor1`.

Returns :

a `YGenericSensor` object allowing you to drive the generic sensor.

YGenericSensor.FirstGenericSensor() YGenericSensor

YGenericSensor.FirstGenericSensor()

Starts the enumeration of generic sensors currently accessible.

js	function **yFirstGenericSensor**( )
cpp	YGenericSensor * **yFirstGenericSensor**( )
m	+(YGenericSensor*) **FirstGenericSensor**
pas	TYGenericSensor **yFirstGenericSensor**( ): TYGenericSensor
vb	function **yFirstGenericSensor**( ) As YGenericSensor
cs	static YGenericSensor **FirstGenericSensor**( )
java	static YGenericSensor **FirstGenericSensor**( )
uwp	static YGenericSensor **FirstGenericSensor**( )
py	**FirstGenericSensor**( )
php	function **yFirstGenericSensor**( )
es	static **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.

YGenericSensor.FirstGenericSensorInContext() YGenericSensor

YGenericSensor.FirstGenericSensorInContext()

Starts the enumeration of generic sensors currently accessible.

java	static YGenericSensor **FirstGenericSensorInContext**(YAPIContext **yctx**)
uwp	static YGenericSensor **FirstGenericSensorInContext**(YAPIContext **yctx**)
es	static **FirstGenericSensorInContext**(**yctx**)

Use the method `YGenericSensor.nextGenericSensor()` to iterate on next generic sensors.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YGenericSensor` object, corresponding to the first generic sensor currently online, or a `null` pointer if there are none.

YGenericSensor.GetSimilarFunctions() YGenericSensor

YGenericSensor.GetSimilarFunctions()

Enumerates all functions of type GenericSensor available on the devices currently reachable by the library, and returns their unique hardware ID.

dnp static new string[] **GetSimilarFunctions**( )

cp static vector<string> **GetSimilarFunctions**( )

Each of these IDs can be provided as argument to the method `YGenericSensor.FindGenericSensor` to obtain an object that can control the corresponding device.

Returns :

an array of strings, each string containing the unique hardwareId of a device function currently connected.

genericsensor→AdvMode YGenericSensor

Measuring mode used for the advertised value pushed to the parent hub.

dnp int **AdvMode**

Possible values:

```
Y_ADVMODE_INVALID    = 0
Y_ADVMODE_IMMEDIATE  = 1
Y_ADVMODE_PERIOD_AVG = 2
Y_ADVMODE_PERIOD_MIN = 3
Y_ADVMODE_PERIOD_MAX = 4
```

Writable. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

genericsensor→AdvertisedValue YGenericSensor

Short string representing the current state of the function.

dnp string **AdvertisedValue**

genericsensor→Enabled YGenericSensor

Activation state of this input.

dnp int **Enabled**

Possible values:

Y_ENABLED_INVALID = 0
Y_ENABLED_FALSE = 1
Y_ENABLED_TRUE = 2

Writable. When an input is disabled, its value is no more updated. On some devices, disabling an input can improve the refresh rate of the other active inputs. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

genericsensor→FriendlyName YGenericSensor

Global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

dnp string **FriendlyName**

The returned string uses the logical names of the module and of the function if they are defined, otherwise the serial number of the module and the hardware identifier of the function (for example: `MyCustomName.relay1`)

genericsensor→FunctionId YGenericSensor

Hardware identifier of the sensor, without reference to the module.

dnp string **FunctionId**

For example `relay1`

genericsensor→HardwareId YGenericSensor

Unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`.

dnp string **HardwareId**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example `RELAYLO1-123456.relay1`).

genericsensor→IsOnline YGenericSensor

Checks if the function is currently reachable.

dnp bool **IsOnline**

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the function.

genericsensor→LogFrequency YGenericSensor

Datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

dnp string **LogFrequency**

Writable. Changes the datalogger recording frequency for this function. The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF". Note that setting the datalogger recording frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

genericsensor→LogicalName YGenericSensor

Logical name of the function.

dnp string **LogicalName**

Writable. You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

genericsensor→ReportFrequency YGenericSensor

Timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

dnp string **ReportFrequency**

Writable. Changes the timed value notification frequency for this function. The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (e.g. "4/h"). To disable timed value notifications for this function, use the value "OFF". Note that setting the timed value notification frequency to a greater value than the sensor native sampling frequency is unless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

genericsensor→Resolution YGenericSensor

Resolution of the measured values.

dnp double **Resolution**

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Writable. Changes the resolution of the measured physical values. The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

genericsensor→SerialNumber YGenericSensor

Serial number of the module, as set by the factory.

dnp string **SerialNumber**

genericsensor→SignalBias YGenericSensor

Electric signal bias for zero shift adjustment.

dnp double **SignalBias**

A positive bias means that the signal is over-reporting the measure, while a negative bias means that the signal is under-reporting the measure.

Writable. If your electric signal reads positive when it should be zero, setup a positive signalBias of the same value to fix the zero shift. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

genericsensor→SignalRange YGenericSensor

Input signal range used by the sensor.

dnp string **SignalRange**

Writable. When the input signal gets out of the planned range, the output value will be set to an arbitrary large value, whose sign indicates the direction of the range overrun.

For a 4-20mA sensor, the default input signal range is "4...20". For a 0-10V sensor, the default input signal range is "0.1...10". For numeric communication interfaces, the default input signal range is "-999999.999...999999.999".

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

genericsensor→SignalSampling YGenericSensor

Electric signal sampling method to use.

dnp int **SignalSampling**

The `HIGH_RATE` method uses the highest sampling frequency, without any filtering. The `HIGH_RATE_FILTERED` method adds a windowed 7-sample median filter. The `LOW_NOISE` method uses a reduced acquisition frequency to reduce noise. The `LOW_NOISE_FILTERED` method combines a reduced frequency with the median filter to get measures as stable as possible when working on a noisy signal.

Possible values:

```
Y_SIGNALSAMPLING_INVALID            = 0
Y_SIGNALSAMPLING_HIGH_RATE          = 1
Y_SIGNALSAMPLING_HIGH_RATE_FILTERED = 2
Y_SIGNALSAMPLING_LOW_NOISE          = 3
Y_SIGNALSAMPLING_LOW_NOISE_FILTERED = 4
Y_SIGNALSAMPLING_HIGHEST_RATE       = 5
```

Writable. The `HIGH_RATE` method uses the highest sampling frequency, without any filtering. The `HIGH_RATE_FILTERED` method adds a windowed 7-sample median filter. The `LOW_NOISE` method uses a reduced acquisition frequency to reduce noise. The `LOW_NOISE_FILTERED` method combines a reduced frequency with the median filter to get measures as stable as possible when working on a noisy signal. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

genericsensor→SignalUnit YGenericSensor

Measuring unit of the electrical signal used by the sensor.

dnp string **SignalUnit**

genericsensor→ValueRange YGenericSensor

Physical value range measured by the sensor.

dnp string **ValueRange**

Writable. Changes the output value range, corresponding to the physical value measured by the sensor. The default output value range is the same as the input signal range (1:1 mapping), but you can change it so that the function automatically computes the physical value encoded by the input signal. Be aware that, as a side effect, the range modification may automatically modify the display resolution.

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

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)
cpp  int calibrateFromPoints( vector<double> rawValues,
                              vector<double> refValues)
m    -(int) calibrateFromPoints : (NSMutableArray*) rawValues
                                : (NSMutableArray*) refValues
pas  LongInt calibrateFromPoints( rawValues: TDoubleArray,
                                  refValues: TDoubleArray): LongInt
vb   procedure calibrateFromPoints( ByVal rawValues As List(Of)
cs   int calibrateFromPoints( List<double> rawValues,
                              List<double> refValues)
java int calibrateFromPoints( ArrayList<Double> rawValues,
                              ArrayList<Double> refValues)
uwp  async Task<int> calibrateFromPoints( List<double> rawValues,
                                          List<double> refValues)
py   calibrateFromPoints( rawValues, refValues)
php  function calibrateFromPoints( $rawValues, $refValues)
es   async calibrateFromPoints( rawValues, refValues)
dnp  int calibrateFromPoints( double[] rawValues,
                              double[] refValues)
cp   int calibrateFromPoints( vector<double> rawValues,
                              vector<double> 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 values for the correction points.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→clearCache() YGenericSensor

Invalidates the cache.

js	function **clearCache**( )
cpp	void **clearCache**( )
m	-(void) **clearCache**
pas	**clearCache**( )
vb	procedure **clearCache**( )
cs	void **clearCache**( )
java	void **clearCache**( )
py	**clearCache**( )
php	function **clearCache**( )
es	async **clearCache**( )

Invalidates the cache of the generic sensor attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

genericsensor→describe() YGenericSensor

Returns a short text that describes unambiguously the instance of the generic sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

js	function **describe**( )
cpp	string **describe**( )
m	-(NSString*) **describe**
pas	string **describe**( ): string
vb	function **describe**( ) As String
cs	string **describe**( )
java	String **describe**( )
py	**describe**( )
php	function **describe**( )
es	async **describe**( )

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :
a string that describes the generic sensor (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

genericsensor→get_advMode() genericsensor→advMode()

YGenericSensor

Returns the measuring mode used for the advertised value pushed to the parent hub.

js	function **get_advMode**( )
cpp	Y_ADVMODE_enum **get_advMode**( )
m	-(Y_ADVMODE_enum) advMode
pas	Integer **get_advMode**( ): Integer
vb	function **get_advMode**( ) As Integer
cs	int **get_advMode**( )
java	int **get_advMode**( )
uwp	async Task<int> **get_advMode**( )
py	**get_advMode**( )
php	function **get_advMode**( )
es	async **get_advMode**( )
dnp	int **get_advMode**( )
cp	int **get_advMode**( )
cmd	YGenericSensor **target get_advMode**

Returns :

a value among `Y_ADVMODE_IMMEDIATE`, `Y_ADVMODE_PERIOD_AVG`, `Y_ADVMODE_PERIOD_MIN` and `Y_ADVMODE_PERIOD_MAX` corresponding to the measuring mode used for the advertised value pushed to the parent hub

On failure, throws an exception or returns `Y_ADVMODE_INVALID`.

genericsensor→get_advertisedValue() genericsensor→advertisedValue() YGenericSensor

Returns the current value of the generic sensor (no more than 6 characters).

```
js    function get_advertisedValue( )
cpp   string get_advertisedValue( )
m     -(NSString*) advertisedValue
pas   string get_advertisedValue( ): string
vb    function get_advertisedValue( ) As String
cs    string get_advertisedValue( )
java  String get_advertisedValue( )
uwp   async Task<string> get_advertisedValue( )
py    get_advertisedValue( )
php   function get_advertisedValue( )
es    async get_advertisedValue( )
dnp   string get_advertisedValue( )
cp    string 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**( )
cpp	double **get_currentRawValue**( )
m	-(double) currentRawValue
pas	double **get_currentRawValue**( ): double
vb	function **get_currentRawValue**( ) As Double
cs	double **get_currentRawValue**( )
java	double **get_currentRawValue**( )
uwp	async Task<double> **get_currentRawValue**( )
py	**get_currentRawValue**( )
php	function **get_currentRawValue**( )
es	async **get_currentRawValue**( )
dnp	double **get_currentRawValue**( )
cp	double **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( )
cpp   double get_currentValue( )
m     -(double) currentValue
pas   double get_currentValue( ): double
vb    function get_currentValue( ) As Double
cs    double get_currentValue( )
java  double get_currentValue( )
uwp   async Task<double> get_currentValue( )
py    get_currentValue( )
php   function get_currentValue( )
es    async get_currentValue( )
dnp   double get_currentValue( )
cp    double 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_dataLogger() genericsensor→dataLogger()

YGenericSensor

Returns the `YDatalogger` object of the device hosting the sensor.

js	function **get_dataLogger**( )
cpp	YDataLogger* **get_dataLogger**( )
m	-(YDataLogger*) dataLogger
pas	TYDataLogger **get_dataLogger**( ): TYDataLogger
vb	function **get_dataLogger**( ) As YDataLogger
cs	YDataLogger **get_dataLogger**( )
java	YDataLogger **get_dataLogger**( )
uwp	async Task<YDataLogger> **get_dataLogger**( )
py	**get_dataLogger**( )
php	function **get_dataLogger**( )
es	async **get_dataLogger**( )
dnp	YDataLoggerProxy **get_dataLogger**( )
cp	YDataLoggerProxy* **get_dataLogger**( )

This method returns an object that can control global parameters of the data logger. The returned object should not be freed.

Returns :
an `YDatalogger` object, or null on error.

genericsensor→get_enabled() genericsensor→enabled() YGenericSensor

Returns the activation state of this input.

js	function **get_enabled**( )
cpp	Y_ENABLED_enum **get_enabled**( )
m	-(Y_ENABLED_enum) enabled
pas	Integer **get_enabled**( ): Integer
vb	function **get_enabled**( ) As Integer
cs	int **get_enabled**( )
java	int **get_enabled**( )
uwp	async Task<int> **get_enabled**( )
py	**get_enabled**( )
php	function **get_enabled**( )
es	async **get_enabled**( )
dnp	int **get_enabled**( )
cp	int **get_enabled**( )
cmd	YGenericSensor **target get_enabled**

Returns :
either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`, according to the activation state of this input

On failure, throws an exception or returns `Y_ENABLED_INVALID`.

genericsensor→get_errorMessage() genericsensor→errorMessage()

YGenericSensor

Returns the error message of the latest error with the generic sensor.

js	function **get_errorMessage**( )
cpp	string **get_errorMessage**( )
m	-(NSString*) errorMessage
pas	string **get_errorMessage**( ): string
vb	function **get_errorMessage**( ) As String
cs	string **get_errorMessage**( )
java	String **get_errorMessage**( )
py	**get_errorMessage**( )
php	function **get_errorMessage**( )
es	**get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the 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( )
cpp   YRETCODE get_errorType( )
m     -(YRETCODE) errorType
pas   YRETCODE get_errorType( ): YRETCODE
vb    function get_errorType( ) As YRETCODE
cs    YRETCODE get_errorType( )
java  int get_errorType( )
py    get_errorType( )
php   function get_errorType( )
es    get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the 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**( )
cpp	string **get_friendlyName**( )
m	-(NSString*) friendlyName
cs	string **get_friendlyName**( )
java	String **get_friendlyName**( )
py	**get_friendlyName**( )
php	function **get_friendlyName**( )
es	async **get_friendlyName**( )
dnp	string **get_friendlyName**( )
cp	string **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**( )
cpp	YFUN_DESCR **get_functionDescriptor**( )
m	-(YFUN_DESCR) functionDescriptor
pas	YFUN_DESCR **get_functionDescriptor**( ): YFUN_DESCR
vb	function **get_functionDescriptor**( ) As YFUN_DESCR
cs	YFUN_DESCR **get_functionDescriptor**( )
java	String **get_functionDescriptor**( )
py	**get_functionDescriptor**( )
php	function **get_functionDescriptor**( )
es	async **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :
an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

genericsensor→get_functionId() genericsensor→functionId()

YGenericSensor

Returns the hardware identifier of the generic sensor, without reference to the module.

js	function **get_functionId**( )
cpp	string **get_functionId**( )
m	-(NSString*) functionId
vb	function **get_functionId**( ) As String
cs	string **get_functionId**( )
java	String **get_functionId**( )
py	**get_functionId**( )
php	function **get_functionId**( )
es	async **get_functionId**( )
dnp	string **get_functionId**( )
cp	string **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**( )
cpp	string **get_hardwareId**( )
m	-(NSString*) hardwareId
vb	function **get_hardwareId**( ) As String
cs	string **get_hardwareId**( )
java	String **get_hardwareId**( )
py	**get_hardwareId**( )
php	function **get_hardwareId**( )
es	async **get_hardwareId**( )
dnp	string **get_hardwareId**( )
cp	string **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 for the measure since the device was started.

```
js    function get_highestValue( )
cpp   double get_highestValue( )
m     -(double) highestValue
pas   double get_highestValue( ): double
vb    function get_highestValue( ) As Double
cs    double get_highestValue( )
java  double get_highestValue( )
uwp   async Task<double> get_highestValue( )
py    get_highestValue( )
php   function get_highestValue( )
es    async get_highestValue( )
dnp   double get_highestValue( )
cp    double get_highestValue( )
cmd   YGenericSensor target get_highestValue
```

Can be reset to an arbitrary value thanks to set_highestValue().

Returns :

a floating point number corresponding to the maximal value observed for the measure since the device was started

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( )
cpp   string get_logFrequency( )
m     -(NSString*) logFrequency
pas   string get_logFrequency( ): string
vb    function get_logFrequency( ) As String
cs    string get_logFrequency( )
java  String get_logFrequency( )
uwp   async Task<string> get_logFrequency( )
py    get_logFrequency( )
php   function get_logFrequency( )
es    async get_logFrequency( )
dnp   string get_logFrequency( )
cp    string 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**( )
cpp	string **get_logicalName**( )
m	-(NSString*) logicalName
pas	string **get_logicalName**( ): string
vb	function **get_logicalName**( ) As String
cs	string **get_logicalName**( )
java	String **get_logicalName**( )
uwp	async Task<string> **get_logicalName**( )
py	**get_logicalName**( )
php	function **get_logicalName**( )
es	async **get_logicalName**( )
dnp	string **get_logicalName**( )
cp	string **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 for the measure since the device was started.

```
js    function get_lowestValue( )
cpp   double get_lowestValue( )
m     -(double) lowestValue
pas   double get_lowestValue( ): double
vb    function get_lowestValue( ) As Double
cs    double get_lowestValue( )
java  double get_lowestValue( )
uwp   async Task<double> get_lowestValue( )
py    get_lowestValue( )
php   function get_lowestValue( )
es    async get_lowestValue( )
dnp   double get_lowestValue( )
cp    double get_lowestValue( )
cmd   YGenericSensor target get_lowestValue
```

Can be reset to an arbitrary value thanks to set_lowestValue().

Returns :

a floating point number corresponding to the minimal value observed for the measure since the device was started

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**( )
cpp	YModule * **get_module**( )
m	-(YModule*) module
pas	TYModule **get_module**( ): TYModule
vb	function **get_module**( ) As YModule
cs	YModule **get_module**( )
java	YModule **get_module**( )
py	**get_module**( )
php	function **get_module**( )
es	async **get_module**( )
dnp	YModuleProxy **get_module**( )
cp	YModuleProxy * **get_module**( )

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

Returns :
an instance of `YModule`

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)
```

If the function cannot be located on any module, the returned `YModule` object does not show as online.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking Firefox JavaScript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous JavasSript calls for more details.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

genericsensor→get_recordedData() genericsensor→recordedData() YGenericSensor

Retrieves a `YDataSet` object holding historical data for this sensor, for a specified time interval.

```
js    function get_recordedData( startTime, endTime)
cpp   YDataSet get_recordedData( double startTime, double endTime)
m     -(YDataSet*) recordedData : (double) startTime
                                : (double) endTime
pas   TYDataSet get_recordedData( startTime: double, endTime: double): TYDataSet
vb    function get_recordedData( ByVal startTime As Double,
                                 ByVal endTime As Double) As YDataSet
cs    YDataSet get_recordedData( double startTime, double endTime)
java  YDataSet get_recordedData( double startTime, double endTime)
uwp   async Task<YDataSet> get_recordedData( double startTime,
                                             double endTime)
py    get_recordedData( startTime, endTime)
php   function get_recordedData( $startTime, $endTime)
es    async get_recordedData( startTime, endTime)
dnp   YDataSetProxy get_recordedData( double startTime,
                                      double endTime)
cp    YDataSetProxy* get_recordedData( double startTime,
                                       double 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 `YDataSet` class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as `YDataSet` objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without ending limit.

Returns :

an instance of `YDataSet`, providing access to historical data. Past measures can be loaded progressively using methods from the `YDataSet` object.

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**( )
cpp	string **get_reportFrequency**( )
m	-(NSString*) reportFrequency
pas	string **get_reportFrequency**( ): string
vb	function **get_reportFrequency**( ) As String
cs	string **get_reportFrequency**( )
java	String **get_reportFrequency**( )
uwp	async Task<string> **get_reportFrequency**( )
py	**get_reportFrequency**( )
php	function **get_reportFrequency**( )
es	async **get_reportFrequency**( )
dnp	string **get_reportFrequency**( )
cp	string **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**( )
cpp	double **get_resolution**( )
m	-(double) resolution
pas	double **get_resolution**( ): double
vb	function **get_resolution**( ) As Double
cs	double **get_resolution**( )
java	double **get_resolution**( )
uwp	async Task<double> **get_resolution**( )
py	**get_resolution**( )
php	function **get_resolution**( )
es	async **get_resolution**( )
dnp	double **get_resolution**( )
cp	double **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. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `Y_RESOLUTION_INVALID`.

genericsensor→get_sensorState() genericsensor→sensorState()

YGenericSensor

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

js	function **get_sensorState**( )
cpp	int **get_sensorState**( )
m	-(int) sensorState
pas	LongInt **get_sensorState**( ): LongInt
vb	function **get_sensorState**( ) As Integer
cs	int **get_sensorState**( )
java	int **get_sensorState**( )
uwp	async Task<int> **get_sensorState**( )
py	**get_sensorState**( )
php	function **get_sensorState**( )
es	async **get_sensorState**( )
dnp	int **get_sensorState**( )
cp	int **get_sensorState**( )
cmd	YGenericSensor **target get_sensorState**

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `Y_SENSORSTATE_INVALID`.

genericsensor→get_serialNumber() genericsensor→serialNumber()

YGenericSensor

Returns the serial number of the module, as set by the factory.

js	function **get_serialNumber**( )
cpp	string **get_serialNumber**( )
m	-(NSString*) serialNumber
pas	string **get_serialNumber**( ): string
vb	function **get_serialNumber**( ) As String
cs	string **get_serialNumber**( )
java	String **get_serialNumber**( )
uwp	async Task<string> **get_serialNumber**( )
py	**get_serialNumber**( )
php	function **get_serialNumber**( )
es	async **get_serialNumber**( )
dnp	string **get_serialNumber**( )
cp	string **get_serialNumber**( )
cmd	YGenericSensor **target get_serialNumber**

Returns :

a string corresponding to the serial number of the module, as set by the factory.

On failure, throws an exception or returns YModule.SERIALNUMBER_INVALID.

genericsensor→get_signalBias() genericsensor→signalBias() YGenericSensor

Returns the electric signal bias for zero shift adjustment.

```
js    function get_signalBias( )
cpp   double get_signalBias( )
m     -(double) signalBias
pas   double get_signalBias( ): double
vb    function get_signalBias( ) As Double
cs    double get_signalBias( )
java  double get_signalBias( )
uwp   async Task<double> get_signalBias( )
py    get_signalBias( )
php   function get_signalBias( )
es    async get_signalBias( )
dnp   double get_signalBias( )
cp    double get_signalBias( )
cmd   YGenericSensor target get_signalBias
```

A positive bias means that the signal is over-reporting the measure, while a negative bias means that the signal is under-reporting the measure.

Returns :
a floating point number corresponding to the electric signal bias for zero shift adjustment

On failure, throws an exception or returns `Y_SIGNALBIAS_INVALID`.

genericsensor→get_signalRange() genericsensor→signalRange()

YGenericSensor

Returns the input signal range used by the sensor.

js	function **get_signalRange**( )
cpp	string **get_signalRange**( )
m	-(NSString*) signalRange
pas	string **get_signalRange**( ): string
vb	function **get_signalRange**( ) As String
cs	string **get_signalRange**( )
java	String **get_signalRange**( )
uwp	async Task<string> **get_signalRange**( )
py	**get_signalRange**( )
php	function **get_signalRange**( )
es	async **get_signalRange**( )
dnp	string **get_signalRange**( )
cp	string **get_signalRange**( )
cmd	YGenericSensor **target get_signalRange**

Returns :

a string corresponding to the input signal range used by the sensor

On failure, throws an exception or returns `Y_SIGNALRANGE_INVALID`.

genericsensor→get_signalSampling() genericsensor→signalSampling() YGenericSensor

Returns the electric signal sampling method to use.

js	function **get_signalSampling**( )
cpp	Y_SIGNALSAMPLING_enum **get_signalSampling**( )
m	-(Y_SIGNALSAMPLING_enum) signalSampling
pas	Integer **get_signalSampling**( ): Integer
vb	function **get_signalSampling**( ) As Integer
cs	int **get_signalSampling**( )
java	int **get_signalSampling**( )
uwp	async Task<int> **get_signalSampling**( )
py	**get_signalSampling**( )
php	function **get_signalSampling**( )
es	async **get_signalSampling**( )
dnp	int **get_signalSampling**( )
cp	int **get_signalSampling**( )
cmd	YGenericSensor **target get_signalSampling**

The `HIGH_RATE` method uses the highest sampling frequency, without any filtering. The `HIGH_RATE_FILTERED` method adds a windowed 7-sample median filter. The `LOW_NOISE` method uses a reduced acquisition frequency to reduce noise. The `LOW_NOISE_FILTERED` method combines a reduced frequency with the median filter to get measures as stable as possible when working on a noisy signal.

Returns :

a value among `Y_SIGNALSAMPLING_HIGH_RATE`, `Y_SIGNALSAMPLING_HIGH_RATE_FILTERED`, `Y_SIGNALSAMPLING_LOW_NOISE`, `Y_SIGNALSAMPLING_LOW_NOISE_FILTERED` and `Y_SIGNALSAMPLING_HIGHEST_RATE` corresponding to the electric signal sampling method to use

On failure, throws an exception or returns `Y_SIGNALSAMPLING_INVALID`.

genericsensor→get_signalUnit() genericsensor→signalUnit() YGenericSensor

Returns the measuring unit of the electrical signal used by the sensor.

```
js    function get_signalUnit( )
cpp   string get_signalUnit( )
m     -(NSString*) signalUnit
pas   string get_signalUnit( ): string
vb    function get_signalUnit( ) As String
cs    string get_signalUnit( )
java  String get_signalUnit( )
uwp   async Task<string> get_signalUnit( )
py    get_signalUnit( )
php   function get_signalUnit( )
es    async get_signalUnit( )
dnp   string get_signalUnit( )
cp    string 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 current value of the electrical signal measured by the sensor.

js	function **get_signalValue**( )
cpp	double **get_signalValue**( )
m	-(double) signalValue
pas	double **get_signalValue**( ): double
vb	function **get_signalValue**( ) As Double
cs	double **get_signalValue**( )
java	double **get_signalValue**( )
uwp	async Task<double> **get_signalValue**( )
py	**get_signalValue**( )
php	function **get_signalValue**( )
es	async **get_signalValue**( )
dnp	double **get_signalValue**( )
cp	double **get_signalValue**( )
cmd	YGenericSensor **target get_signalValue**

Returns :

a floating point number corresponding to the current value of the electrical signal measured 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 measure.

js	function **get_unit**( )
cpp	string **get_unit**( )
m	-(NSString*) unit
pas	string **get_unit**( ): string
vb	function **get_unit**( ) As String
cs	string **get_unit**( )
java	String **get_unit**( )
uwp	async Task<string> **get_unit**( )
py	**get_unit**( )
php	function **get_unit**( )
es	async **get_unit**( )
dnp	string **get_unit**( )
cp	string **get_unit**( )
cmd	YGenericSensor **target get_unit**

Returns :

a string corresponding to the measuring unit for the measure

On failure, throws an exception or returns `Y_UNIT_INVALID`.

genericsensor→get_userData() genericsensor→userData() YGenericSensor

Returns the value of the userData attribute, as previously stored using method `set_userData`.

js	function **get_userData**( )
cpp	void * **get_userData**( )
m	-(id) userData
pas	Tobject **get_userData**( ): Tobject
vb	function **get_userData**( ) As Object
cs	object **get_userData**( )
java	Object **get_userData**( )
py	**get_userData**( )
php	function **get_userData**( )
es	async **get_userData**( )

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

genericsensor→get_valueRange() genericsensor→valueRange()

YGenericSensor

Returns the physical value range measured by the sensor.

js	function **get_valueRange**( )
cpp	string **get_valueRange**( )
m	-(NSString*) valueRange
pas	string **get_valueRange**( ): string
vb	function **get_valueRange**( ) As String
cs	string **get_valueRange**( )
java	String **get_valueRange**( )
uwp	async Task<string> **get_valueRange**( )
py	**get_valueRange**( )
php	function **get_valueRange**( )
es	async **get_valueRange**( )
dnp	string **get_valueRange**( )
cp	string **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**( )
cpp	bool **isOnline**( )
m	-(BOOL) **isOnline**
pas	boolean **isOnline**( ): boolean
vb	function **isOnline**( ) As Boolean
cs	bool **isOnline**( )
java	boolean **isOnline**( )
py	**isOnline**( )
php	function **isOnline**( )
es	async **isOnline**( )
dnp	bool **isOnline**( )
cp	bool **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)
```

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→isReadOnly() YGenericSensor

Test if the function is readOnly.

cpp	bool **isReadOnly**( )
m	-(bool) **isReadOnly**
pas	boolean **isReadOnly**( ): boolean
vb	function **isReadOnly**( ) As Boolean
cs	bool **isReadOnly**( )
java	boolean **isReadOnly**( )
uwp	async Task<bool> **isReadOnly**( )
py	**isReadOnly**( )
php	function **isReadOnly**( )
es	async **isReadOnly**( )
dnp	bool **isReadOnly**( )
cp	bool **isReadOnly**( )
cmd	YGenericSensor **target isReadOnly**

Return `true` if the function is write protected or that the function is not available.

Returns :
`true` if the function is readOnly or not online.

genericsensor→isSensorReady() YGenericSensor

Checks if the sensor is currently able to provide an up-to-date measure.

cmd YGenericSensor **target isSensorReady**

Returns false if the device is unreachable, or if the sensor does not have a current measure to transmit. No exception is raised if there is an error while trying to contact the device hosting $THEFUNCTION$.

Returns :
`true` if the sensor can provide an up-to-date measure, and `false` otherwise

genericsensor→load() YGenericSensor

Preloads the generic sensor cache with a specified validity duration.

```
js    function load( msValidity)
cpp   YRETCODE load( int msValidity)
m     -(YRETCODE) load : (u64) msValidity
pas   YRETCODE load( msValidity: u64): YRETCODE
vb    function load( ByVal msValidity As Long) As YRETCODE
cs    YRETCODE load( ulong msValidity)
java  int load( long msValidity)
py    load( msValidity)
php   function load( $msValidity)
es    async load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→loadAttribute() YGenericSensor

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
js    function loadAttribute( attrName)
cpp   string loadAttribute( string attrName)
m     -(NSString*) loadAttribute : (NSString*) attrName
pas   string loadAttribute( attrName: string): string
vb    function loadAttribute( ByVal attrName As String) As String
cs    string loadAttribute( string attrName)
java  String loadAttribute( String attrName)
uwp   async Task<string> loadAttribute( string attrName)
py    loadAttribute( attrName)
php   function loadAttribute( $attrName)
es    async loadAttribute( attrName)
dnp   string loadAttribute( string attrName)
cp    string loadAttribute( string attrName)
```

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

genericsensor→loadCalibrationPoints() YGenericSensor

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```
js   function loadCalibrationPoints( rawValues, refValues)
cpp  int loadCalibrationPoints( vector<double> rawValues,
                                vector<double> refValues)
m    -(int) loadCalibrationPoints : (NSMutableArray*) rawValues
                                  : (NSMutableArray*) refValues
pas  LongInt loadCalibrationPoints( var rawValues: TDoubleArray,
                                    var refValues: TDoubleArray): LongInt
vb   procedure loadCalibrationPoints( ByVal rawValues As List(Of)
cs   int loadCalibrationPoints( List<double> rawValues,
                                List<double> refValues)
java int loadCalibrationPoints( ArrayList<Double> rawValues,
                                ArrayList<Double> refValues)
uwp  async Task<int> loadCalibrationPoints( List<double> rawValues,
                                            List<double> refValues)
py   loadCalibrationPoints( rawValues, refValues)
php  function loadCalibrationPoints( &$rawValues, &$refValues)
es   async loadCalibrationPoints( rawValues, refValues)
cmd  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)
```

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→muteValueCallbacks() YGenericSensor

Disables the propagation of every new advertised value to the parent hub.

js	function **muteValueCallbacks**( )
cpp	int **muteValueCallbacks**( )
m	-(int) **muteValueCallbacks**
pas	LongInt **muteValueCallbacks**( ): LongInt
vb	function **muteValueCallbacks**( ) As Integer
cs	int **muteValueCallbacks**( )
java	int **muteValueCallbacks**( )
uwp	async Task<int> **muteValueCallbacks**( )
py	**muteValueCallbacks**( )
php	function **muteValueCallbacks**( )
es	async **muteValueCallbacks**( )
dnp	int **muteValueCallbacks**( )
cp	int **muteValueCallbacks**( )
cmd	YGenericSensor **target muteValueCallbacks**

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→nextGenericSensor() YGenericSensor

Continues the enumeration of generic sensors started using `yFirstGenericSensor()`.

js	function **nextGenericSensor**( )
cpp	YGenericSensor * **nextGenericSensor**( )
m	-(YGenericSensor*) **nextGenericSensor**
pas	TYGenericSensor **nextGenericSensor**( ): TYGenericSensor
vb	function **nextGenericSensor**( ) As YGenericSensor
cs	YGenericSensor **nextGenericSensor**( )
java	YGenericSensor **nextGenericSensor**( )
uwp	YGenericSensor **nextGenericSensor**( )
py	**nextGenericSensor**( )
php	function **nextGenericSensor**( )
es	**nextGenericSensor**( )

Caution: You can't make any assumption about the returned generic sensors order. If you want to find a specific a generic sensor, use `GenericSensor.findGenericSensor()` and a hardwareID or a logical name.

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**)
cpp	int **registerTimedReportCallback**(YGenericSensorTimedReportCallback **callback**)
m	-(int) **registerTimedReportCallback** : (YGenericSensorTimedReportCallback) **callback**
pas	LongInt **registerTimedReportCallback**(**callback**: TYGenericSensorTimedReportCallback): LongInt
vb	function **registerTimedReportCallback**(ByVal **callback** As YGenericSensorTimedReportCallback) As Integer
cs	int **registerTimedReportCallback**(TimedReportCallback **callback**)
java	int **registerTimedReportCallback**(TimedReportCallback **callback**)
uwp	async Task<int> **registerTimedReportCallback**(TimedReportCallback **callback**)
py	**registerTimedReportCallback**(**callback**)
php	function **registerTimedReportCallback**(**$callback**)
es	async **registerTimedReportCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

genericsensor→registerValueCallback() YGenericSensor

Registers the callback function that is invoked on every change of advertised value.

js	function **registerValueCallback**(**callback**)
cpp	int **registerValueCallback**(YGenericSensorValueCallback **callback**)
m	-(int) **registerValueCallback** : (YGenericSensorValueCallback) **callback**
pas	LongInt **registerValueCallback**(**callback**: TYGenericSensorValueCallback): LongInt
vb	function **registerValueCallback**(ByVal **callback** As YGenericSensorValueCallback) As Integer
cs	int **registerValueCallback**(ValueCallback **callback**)
java	int **registerValueCallback**(UpdateCallback **callback**)
uwp	async Task<int> **registerValueCallback**(ValueCallback **callback**)
py	**registerValueCallback**(**callback**)
php	function **registerValueCallback**(**$callback**)
es	async **registerValueCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

genericsensor→set_advMode() genericsensor→setAdvMode() YGenericSensor

Changes the measuring mode used for the advertised value pushed to the parent hub.

```
js    function set_advMode( newval)
cpp   int set_advMode( Y_ADVMODE_enum newval)
m     -(int) setAdvMode : (Y_ADVMODE_enum) newval
pas   integer set_advMode( newval: Integer): integer
vb    function set_advMode( ByVal newval As Integer) As Integer
cs    int set_advMode( int newval)
java  int set_advMode( int newval)
uwp   async Task<int> set_advMode( int newval)
py    set_advMode( newval)
php   function set_advMode( $newval)
es    async set_advMode( newval)
dnp   int set_advMode( int newval)
cp    int set_advMode( int newval)
cmd   YGenericSensor target set_advMode newval
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a value among `Y_ADVMODE_IMMEDIATE`, `Y_ADVMODE_PERIOD_AVG`, `Y_ADVMODE_PERIOD_MIN` and `Y_ADVMODE_PERIOD_MAX` corresponding to the measuring mode used for the advertised value pushed to the parent hub

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_enabled() genericsensor→setEnabled() YGenericSensor

Changes the activation state of this input.

```
js    function set_enabled( newval)
cpp   int set_enabled( Y_ENABLED_enum newval)
m     -(int) setEnabled : (Y_ENABLED_enum) newval
pas   integer set_enabled( newval: Integer): integer
vb    function set_enabled( ByVal newval As Integer) As Integer
cs    int set_enabled( int newval)
java  int set_enabled( int newval)
uwp   async Task<int> set_enabled( int newval)
py    set_enabled( newval)
php   function set_enabled( $newval)
es    async set_enabled( newval)
dnp   int set_enabled( int newval)
cp    int set_enabled( int newval)
cmd   YGenericSensor target set_enabled newval
```

When an input is disabled, its value is no more updated. On some devices, disabling an input can improve the refresh rate of the other active inputs. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval either `Y_ENABLED_FALSE` or `Y_ENABLED_TRUE`, according to the activation state of this input

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_highestValue() genericsensor→setHighestValue() YGenericSensor

Changes the recorded maximal value observed.

```
js    function set_highestValue( newval)
cpp   int set_highestValue( double newval)
m     -(int) setHighestValue : (double) newval
pas   integer set_highestValue( newval: double): integer
vb    function set_highestValue( ByVal newval As Double) As Integer
cs    int set_highestValue( double newval)
java  int set_highestValue( double newval)
uwp   async Task<int> set_highestValue( double newval)
py    set_highestValue( newval)
php   function set_highestValue( $newval)
es    async set_highestValue( newval)
dnp   int set_highestValue( double newval)
cp    int set_highestValue( double newval)
cmd   YGenericSensor target set_highestValue newval
```

Can be used to reset the value returned by get_lowestValue().

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_logFrequency() genericsensor→setLogFrequency() YGenericSensor

Changes the datalogger recording frequency for this function.

js	function **set_logFrequency**(**newval**)
cpp	int **set_logFrequency**(string **newval**)
m	-(int) setLogFrequency : (NSString*) **newval**
pas	integer **set_logFrequency**(**newval**: string): integer
vb	function **set_logFrequency**(ByVal **newval** As String) As Integer
cs	int **set_logFrequency**(string **newval**)
java	int **set_logFrequency**(String **newval**)
uwp	async Task<int> **set_logFrequency**(string **newval**)
py	**set_logFrequency**(**newval**)
php	function **set_logFrequency**(**$newval**)
es	async **set_logFrequency**(**newval**)
dnp	int **set_logFrequency**(string **newval**)
cp	int **set_logFrequency**(string **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". Note that setting the datalogger recording frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_logicalName() genericsensor→setLogicalName() YGenericSensor

Changes the logical name of the generic sensor.

```
js    function set_logicalName( newval)
cpp   int set_logicalName( string newval)
m     -(int) setLogicalName : (NSString*) newval
pas   integer set_logicalName( newval: string): integer
vb    function set_logicalName( ByVal newval As String) As Integer
cs    int set_logicalName( string newval)
java  int set_logicalName( String newval)
uwp   async Task<int> set_logicalName( string newval)
py    set_logicalName( newval)
php   function set_logicalName( $newval)
es    async set_logicalName( newval)
dnp   int set_logicalName( string newval)
cp    int set_logicalName( string 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)
cpp   int set_lowestValue( double newval)
m     -(int) setLowestValue : (double) newval
pas   integer set_lowestValue( newval: double): integer
vb    function set_lowestValue( ByVal newval As Double) As Integer
cs    int set_lowestValue( double newval)
java  int set_lowestValue( double newval)
uwp   async Task<int> set_lowestValue( double newval)
py    set_lowestValue( newval)
php   function set_lowestValue( $newval)
es    async set_lowestValue( newval)
dnp   int set_lowestValue( double newval)
cp    int set_lowestValue( double newval)
cmd   YGenericSensor target set_lowestValue newval
```

Can be used to reset the value returned by get_lowestValue().

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_reportFrequency() genericsensor→setReportFrequency() YGenericSensor

Changes the timed value notification frequency for this function.

```
js    function set_reportFrequency( newval)
cpp   int set_reportFrequency( string newval)
m     -(int) setReportFrequency : (NSString*) newval
pas   integer set_reportFrequency( newval: string): integer
vb    function set_reportFrequency( ByVal newval As String) As Integer
cs    int set_reportFrequency( string newval)
java  int set_reportFrequency( String newval)
uwp   async Task<int> set_reportFrequency( string newval)
py    set_reportFrequency( newval)
php   function set_reportFrequency( $newval)
es    async set_reportFrequency( newval)
dnp   int set_reportFrequency( string newval)
cp    int set_reportFrequency( string 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 (e.g. "4/h"). To disable timed value notifications for this function, use the value "OFF". Note that setting the timed value notification frequency to a greater value than the sensor native sampling frequency is unless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_resolution() genericsensor→setResolution() YGenericSensor

Modifies the resolution of the measured physical values.

```
js    function set_resolution( newval)
cpp   int set_resolution( double newval)
m     -(int) setResolution : (double) newval
pas   integer set_resolution( newval: double): integer
vb    function set_resolution( ByVal newval As Double) As Integer
cs    int set_resolution( double newval)
java  int set_resolution( double newval)
uwp   async Task<int> set_resolution( double newval)
py    set_resolution( newval)
php   function set_resolution( $newval)
es    async set_resolution( newval)
dnp   int set_resolution( double newval)
cp    int set_resolution( double 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. This feature is very handy when one need to publish values through callbacks with a specific resolution. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

resolution the new resolution as a floating point value such as 1.0, 0.1, 0.01, 0.02 , 0.05 etc.

Returns :

`YAPI_SUCCESS` if the call succeeds. On failure, throws an exception or returns a negative error code.**newval**a floating point number

@return `YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_signalBias() genericsensor→setSignalBias() YGenericSensor

Changes the electric signal bias for zero shift adjustment.

```
js    function set_signalBias( newval)
cpp   int set_signalBias( double newval)
m     -(int) setSignalBias : (double) newval
pas   integer set_signalBias( newval: double): integer
vb    function set_signalBias( ByVal newval As Double) As Integer
cs    int set_signalBias( double newval)
java  int set_signalBias( double newval)
uwp   async Task<int> set_signalBias( double newval)
py    set_signalBias( newval)
php   function set_signalBias( $newval)
es    async set_signalBias( newval)
dnp   int set_signalBias( double newval)
cp    int set_signalBias( double newval)
cmd   YGenericSensor target set_signalBias newval
```

If your electric signal reads positive when it should be zero, setup a positive signalBias of the same value to fix the zero shift. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a floating point number corresponding to the electric signal bias for zero shift adjustment

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→set_signalRange() genericsensor→setSignalRange() YGenericSensor

Changes the input signal range used by the sensor.

```
js    function set_signalRange( newval)
cpp   int set_signalRange( string newval)
m     -(int) setSignalRange : (NSString*) newval
pas   integer 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)
uwp   async Task<int> set_signalRange( string newval)
py    set_signalRange( newval)
php   function set_signalRange( $newval)
es    async set_signalRange( newval)
dnp   int set_signalRange( string newval)
cp    int set_signalRange( string newval)
cmd   YGenericSensor target set_signalRange newval
```

When the input signal gets out of the planned range, the output value will be set to an arbitrary large value, whose sign indicates the direction of the range overrun.

For a 4-20mA sensor, the default input signal range is "4...20". For a 0-10V sensor, the default input signal range is "0.1...10". For numeric communication interfaces, the default input signal range is "-999999.999...999999.999".

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the input 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_signalSampling() genericsensor→setSignalSampling() YGenericSensor

Changes the electric signal sampling method to use.

```
js    function set_signalSampling( newval)
cpp   int set_signalSampling( Y_SIGNALSAMPLING_enum newval)
m     -(int) setSignalSampling : (Y_SIGNALSAMPLING_enum) newval
pas   integer set_signalSampling( newval: Integer): integer
vb    function set_signalSampling( ByVal newval As Integer) As Integer
cs    int set_signalSampling( int newval)
java  int set_signalSampling( int newval)
uwp   async Task<int> set_signalSampling( int newval)
py    set_signalSampling( newval)
php   function set_signalSampling( $newval)
es    async set_signalSampling( newval)
dnp   int set_signalSampling( int newval)
cp    int set_signalSampling( int newval)
cmd   YGenericSensor target set_signalSampling newval
```

The `HIGH_RATE` method uses the highest sampling frequency, without any filtering. The `HIGH_RATE_FILTERED` method adds a windowed 7-sample median filter. The `LOW_NOISE` method uses a reduced acquisition frequency to reduce noise. The `LOW_NOISE_FILTERED` method combines a reduced frequency with the median filter to get measures as stable as possible when working on a noisy signal. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a value among `Y_SIGNALSAMPLING_HIGH_RATE`, `Y_SIGNALSAMPLING_HIGH_RATE_FILTERED`, `Y_SIGNALSAMPLING_LOW_NOISE`, `Y_SIGNALSAMPLING_LOW_NOISE_FILTERED` and `Y_SIGNALSAMPLING_HIGHEST_RATE` corresponding to the electric signal sampling method to use

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)
cpp   int set_unit( string newval)
m     -(int) setUnit : (NSString*) newval
pas   integer 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)
uwp   async Task<int> set_unit( string newval)
py    set_unit( newval)
php   function set_unit( $newval)
es    async set_unit( newval)
dnp   int set_unit( string newval)
cp    int set_unit( string 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)
cpp   void set_userData( void * data)
m     -(void) setUserData : (id) data
pas   set_userData( data: Tobject)
vb    procedure set_userData( ByVal data As Object)
cs    void set_userData( object data)
java  void set_userData( Object data)
py    set_userData( data)
php   function set_userData( $data)
es    async set_userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

genericsensor→set_valueRange() genericsensor→setValueRange() YGenericSensor

Changes the output value range, corresponding to the physical value measured by the sensor.

```
js    function set_valueRange( newval)
cpp   int set_valueRange( string newval)
m     -(int) setValueRange : (NSString*) newval
pas   integer set_valueRange( newval: string): integer
vb    function set_valueRange( ByVal newval As String) As Integer
cs    int set_valueRange( string newval)
java  int set_valueRange( String newval)
uwp   async Task<int> set_valueRange( string newval)
py    set_valueRange( newval)
php   function set_valueRange( $newval)
es    async set_valueRange( newval)
dnp   int set_valueRange( string newval)
cp    int set_valueRange( string newval)
cmd   YGenericSensor target set_valueRange newval
```

The default output value range is the same as the input signal range (1:1 mapping), but you can change it so that the function automatically computes the physical value encoded by the input signal. Be aware that, as a side effect, the range modification may automatically modify the display resolution.

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the output value range, corresponding to the physical value measured by the sensor

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

genericsensor→startDataLogger() YGenericSensor

Starts the data logger on the device.

js	function **startDataLogger**( )
cpp	int **startDataLogger**( )
m	-(int) **startDataLogger**
pas	LongInt **startDataLogger**( ): LongInt
vb	function **startDataLogger**( ) As Integer
cs	int **startDataLogger**( )
java	int **startDataLogger**( )
uwp	async Task<int> **startDataLogger**( )
py	**startDataLogger**( )
php	function **startDataLogger**( )
es	async **startDataLogger**( )
dnp	int **startDataLogger**( )
cp	int **startDataLogger**( )
cmd	YGenericSensor **target startDataLogger**

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :
`YAPI_SUCCESS` if the call succeeds.

genericsensor→stopDataLogger() YGenericSensor

Stops the datalogger on the device.

js	function **stopDataLogger**( )
cpp	int **stopDataLogger**( )
m	-(int) **stopDataLogger**
pas	LongInt **stopDataLogger**( ): LongInt
vb	function **stopDataLogger**( ) As Integer
cs	int **stopDataLogger**( )
java	int **stopDataLogger**( )
uwp	async Task<int> **stopDataLogger**( )
py	**stopDataLogger**( )
php	function **stopDataLogger**( )
es	async **stopDataLogger**( )
dnp	int **stopDataLogger**( )
cp	int **stopDataLogger**( )
cmd	YGenericSensor **target stopDataLogger**

Returns :
`YAPI_SUCCESS` if the call succeeds.

genericsensor→unmuteValueCallbacks() YGenericSensor

Re-enables the propagation of every new advertised value to the parent hub.

js	function **unmuteValueCallbacks**( )
cpp	int **unmuteValueCallbacks**( )
m	-(int) **unmuteValueCallbacks**
pas	LongInt **unmuteValueCallbacks**( ): LongInt
vb	function **unmuteValueCallbacks**( ) As Integer
cs	int **unmuteValueCallbacks**( )
java	int **unmuteValueCallbacks**( )
uwp	async Task<int> **unmuteValueCallbacks**( )
py	**unmuteValueCallbacks**( )
php	function **unmuteValueCallbacks**( )
es	async **unmuteValueCallbacks**( )
dnp	int **unmuteValueCallbacks**( )
cp	int **unmuteValueCallbacks**( )
cmd	YGenericSensor **target unmuteValueCallbacks**

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

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)
es  wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

genericsensor→zeroAdjust() YGenericSensor

Adjusts the signal bias so that the current signal value is need precisely as zero.

js	function **zeroAdjust**( )
cpp	int **zeroAdjust**( )
m	-(int) **zeroAdjust**
pas	LongInt **zeroAdjust**( ): LongInt
vb	function **zeroAdjust**( ) As Integer
cs	int **zeroAdjust**( )
java	int **zeroAdjust**( )
uwp	async Task<int> **zeroAdjust**( )
py	**zeroAdjust**( )
php	function **zeroAdjust**( )
es	async **zeroAdjust**( )
dnp	int **zeroAdjust**( )
cp	int **zeroAdjust**( )
cmd	YGenericSensor **target zeroAdjust**

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

24.6. Class YDataLogger

DataLogger control interface, available on most Yoctopuce sensors.

A non-volatile memory for storing ongoing measured data is available on most Yoctopuce sensors. Recording can happen automatically, without requiring a permanent connection to a computer. The `YDataLogger` class controls the global parameters of the internal data logger. Recording control (start/stop) as well as data retreival is done at sensor objects level.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_module.js'></script>
cpp	#include "yocto_module.h"
m	#import "yocto_module.h"
pas	uses yocto_module;
vb	yocto_module.vb
cs	yocto_module.cs
java	import com.yoctopuce.YoctoAPI.YDataLogger;
uwp	import com.yoctopuce.YoctoAPI.YDataLogger;
py	from yocto_module import *
php	require_once('yocto_module.php');
es	in HTML: <script src="../../lib/yocto_module.js"></script> in node.js: require('yoctolib-es2017/yocto_module.js');
dnp	import YoctoProxyAPI.YDataLoggerProxy
cp	#include "yocto_module_proxy.h"
vi	YDataLogger.vi
ml	import YoctoProxyAPI.YDataLoggerProxy

Global functions
YDataLogger.FindDataLogger(**func**) Retrieves a data logger for a given identifier.
YDataLogger.FindDataLoggerInContext(**yctx**, **func**) Retrieves a data logger for a given identifier in a YAPI context.
YDataLogger.FirstDataLogger() Starts the enumeration of data loggers currently accessible.
YDataLogger.FirstDataLoggerInContext(**yctx**) Starts the enumeration of data loggers currently accessible.
YDataLogger.GetSimilarFunctions() Enumerates all functions of type DataLogger available on the devices currently reachable by the library, and returns their unique hardware ID.
`YDataLogger` properties
datalogger→AdvertisedValue *[read-only]* Short string representing the current state of the function.
datalogger→AutoStart *[writable]* Default activation state of the data logger on power up.
datalogger→BeaconDriven *[writable]* True if the data logger is synchronised with the localization beacon.
datalogger→FriendlyName *[read-only]* Global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

datalogger→FunctionId *[read-only]*

Hardware identifier of the data logger, without reference to the module.

datalogger→HardwareId *[read-only]*

Unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`.

datalogger→IsOnline *[read-only]*

Checks if the function is currently reachable.

datalogger→LogicalName *[writable]*

Logical name of the function.

datalogger→Recording *[writable]*

Current activation state of the data logger.

datalogger→SerialNumber *[read-only]*

Serial number of the module, as set by the factory.

`YDataLogger` methods

datalogger→clearCache()

Invalidates the cache.

datalogger→describe()

Returns a short text that describes unambiguously the instance of the data logger in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

datalogger→forgetAllDataStreams()

Clears the data logger memory and discards all recorded data streams.

datalogger→get_advertisedValue()

Returns the current value of the data logger (no more than 6 characters).

datalogger→get_autoStart()

Returns the default activation state of the data logger on power up.

datalogger→get_beaconDriven()

Returns true if the data logger is synchronised with the localization beacon.

datalogger→get_currentRunIndex()

Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.

datalogger→get_dataSets()

Returns a list of `YDataSet` objects that can be used to retrieve all measures stored by the data logger.

datalogger→get_dataStreams(v)

Builds a list of all data streams hold by the data logger (legacy method).

datalogger→get_errorMessage()

Returns the error message of the latest error with the data logger.

datalogger→get_errorType()

Returns the numerical error code of the latest error with the data logger.

datalogger→get_friendlyName()

Returns a global identifier of the data logger in the format `MODULE_NAME.FUNCTION_NAME`.

datalogger→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

datalogger→get_functionId()

Returns the hardware identifier of the data logger, without reference to the module.

datalogger→get_hardwareId()

Returns the unique hardware identifier of the data logger in the form `SERIAL.FUNCTIONID`.

datalogger→get_logicalName()

Returns the logical name of the data logger.

datalogger→get_module()

Gets the `YModule` object for the device on which the function is located.

datalogger→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

datalogger→get_recording()

Returns the current activation state of the data logger.

datalogger→get_serialNumber()

Returns the serial number of the module, as set by the factory.

datalogger→get_timeUTC()

Returns the Unix timestamp for current UTC time, if known.

datalogger→get_usage()

Returns the percentage of datalogger memory in use.

datalogger→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

datalogger→isOnline()

Checks if the data logger is currently reachable, without raising any error.

datalogger→isOnline_async(callback, context)

Checks if the data logger is currently reachable, without raising any error (asynchronous version).

datalogger→isReadOnly()

Test if the function is readOnly.

datalogger→load(msValidity)

Preloads the data logger cache with a specified validity duration.

datalogger→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

datalogger→load_async(msValidity, callback, context)

Preloads the data logger cache with a specified validity duration (asynchronous version).

datalogger→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

datalogger→nextDataLogger()

Continues the enumeration of data loggers started using `yFirstDataLogger()`.

datalogger→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

datalogger→set_autoStart(newval)

Changes the default activation state of the data logger on power up.

datalogger→set_beaconDriven(newval)

Changes the type of synchronisation of the data logger.

datalogger→set_logicalName(newval)

Changes the logical name of the data logger.

datalogger→set_recording(newval)

Changes the activation state of the data logger to start/stop recording data.

datalogger→set_timeUTC(newval)

Changes the current UTC time reference used for recorded data.

datalogger→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

datalogger→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

datalogger→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YDataLogger.FindDataLogger() YDataLogger.FindDataLogger() — YDataLogger

Retrieves a data logger for a given identifier.

```
js    function yFindDataLogger( func)
cpp   YDataLogger* yFindDataLogger( string func)
m     +(YDataLogger*) FindDataLogger : (NSString*) func
pas   TYDataLogger yFindDataLogger( func: string): TYDataLogger
vb    function yFindDataLogger( ByVal func As String) As YDataLogger
cs    static YDataLogger FindDataLogger( string func)
java  static YDataLogger FindDataLogger( String func)
uwp   static YDataLogger FindDataLogger( string func)
py    FindDataLogger( func)
php   function yFindDataLogger( $func)
es    static FindDataLogger( func)
dnp   static YDataLoggerProxy FindDataLogger( string func)
cp    static YDataLoggerProxy * FindDataLogger( string func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the data logger is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDataLogger.isOnline()` to test if the data logger is indeed online at a given time. In case of ambiguity when looking for a data logger by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the data logger, for instance `Y3DMK002.dataLogger`.

Returns :

a `YDataLogger` object allowing you to drive the data logger.

YDataLogger.FindDataLoggerInContext() YDataLogger

YDataLogger.FindDataLoggerInContext()

Retrieves a data logger for a given identifier in a YAPI context.

```
java  static YDataLogger FindDataLoggerInContext( YAPIContext yctx,
                                                  String func)
uwp   static YDataLogger FindDataLoggerInContext( YAPIContext yctx,
                                                  string func)
es    static FindDataLoggerInContext( yctx, func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the data logger is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDataLogger.isOnline()` to test if the data logger is indeed online at a given time. In case of ambiguity when looking for a data logger by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the data logger, for instance `Y3DMK002.dataLogger`.

Returns :

a `YDataLogger` object allowing you to drive the data logger.

YDataLogger.FirstDataLogger() YDataLogger

YDataLogger.FirstDataLogger()

Starts the enumeration of data loggers currently accessible.

js	function **yFirstDataLogger**( )
cpp	YDataLogger * **yFirstDataLogger**( )
m	+(YDataLogger*) **FirstDataLogger**
pas	TYDataLogger **yFirstDataLogger**( ): TYDataLogger
vb	function **yFirstDataLogger**( ) As YDataLogger
cs	static YDataLogger **FirstDataLogger**( )
java	static YDataLogger **FirstDataLogger**( )
uwp	static YDataLogger **FirstDataLogger**( )
py	**FirstDataLogger**( )
php	function **yFirstDataLogger**( )
es	static **FirstDataLogger**( )

Use the method `YDataLogger.nextDataLogger()` to iterate on next data loggers.

Returns :

a pointer to a `YDataLogger` object, corresponding to the first data logger currently online, or a `null` pointer if there are none.

YDataLogger.FirstDataLoggerInContext() YDataLogger

YDataLogger.FirstDataLoggerInContext()

Starts the enumeration of data loggers currently accessible.

java	static YDataLogger **FirstDataLoggerInContext**(YAPIContext **yctx**)
uwp	static YDataLogger **FirstDataLoggerInContext**(YAPIContext **yctx**)
es	static **FirstDataLoggerInContext**(**yctx**)

Use the method `YDataLogger.nextDataLogger()` to iterate on next data loggers.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YDataLogger` object, corresponding to the first data logger currently online, or a `null` pointer if there are none.

YDataLogger.GetSimilarFunctions() YDataLogger

YDataLogger.GetSimilarFunctions()

Enumerates all functions of type DataLogger available on the devices currently reachable by the library, and returns their unique hardware ID.

dnp static new string[] **GetSimilarFunctions**( )

cp static vector<string> **GetSimilarFunctions**( )

Each of these IDs can be provided as argument to the method `YDataLogger.FindDataLogger` to obtain an object that can control the corresponding device.

Returns :

an array of strings, each string containing the unique hardwareId of a device function currently connected.

datalogger→AdvertisedValue YDataLogger

Short string representing the current state of the function.

dnp string **AdvertisedValue**

datalogger→AutoStart YDataLogger

Default activation state of the data logger on power up.

dnp int **AutoStart**

Possible values:

`Y_AUTOSTART_INVALID`	= 0
`Y_AUTOSTART_OFF`	= 1
`Y_AUTOSTART_ON`	= 2

Writable. Do not forget to call the `saveToFlash()` method of the module to save the configuration change. Note: if the device doesn't have any time source at his disposal when starting up, it will wait for ~8 seconds before automatically starting to record with an arbitrary timestamp

datalogger→BeaconDriven YDataLogger

True if the data logger is synchronised with the localization beacon.

dnp int **BeaconDriven**

Possible values:

```
Y_BEACONDRIVEN_INVALID = 0
Y_BEACONDRIVEN_OFF     = 1
Y_BEACONDRIVEN_ON      = 2
```

Writable. Changes the type of synchronisation of the data logger. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

datalogger→FriendlyName YDataLogger

Global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

dnp string **FriendlyName**

The returned string uses the logical names of the module and of the function if they are defined, otherwise the serial number of the module and the hardware identifier of the function (for example: `MyCustomName.relay1`)

datalogger→FunctionId YDataLogger

Hardware identifier of the data logger, without reference to the module.

dnp string **FunctionId**

For example `relay1`

datalogger→HardwareId YDataLogger

Unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`.

dnp string **HardwareId**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example `RELAYLO1-123456.relay1`).

datalogger→IsOnline YDataLogger

Checks if the function is currently reachable.

dnp bool **IsOnline**

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the function.

datalogger→LogicalName YDataLogger

Logical name of the function.

dnp string **LogicalName**

Writable. You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

datalogger→Recording YDataLogger

Current activation state of the data logger.

dnp int **Recording**

Possible values:

```
Y_RECORDING_INVALID = 0
Y_RECORDING_OFF     = 1
Y_RECORDING_ON      = 2
Y_RECORDING_PENDING = 3
```

Writable. Changes the activation state of the data logger to start/stop recording data.

datalogger→SerialNumber YDataLogger

Serial number of the module, as set by the factory.

dnp string **SerialNumber**

datalogger→clearCache() YDataLogger

Invalidates the cache.

js	function **clearCache**( )
cpp	void **clearCache**( )
m	-(void) **clearCache**
pas	**clearCache**( )
vb	procedure **clearCache**( )
cs	void **clearCache**( )
java	void **clearCache**( )
py	**clearCache**( )
php	function **clearCache**( )
es	async **clearCache**( )

Invalidates the cache of the data logger attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

datalogger→describe() YDataLogger

Returns a short text that describes unambiguously the instance of the data logger in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

```
js    function describe( )
cpp   string describe( )
m     -(NSString*) describe
pas   string describe( ): string
vb    function describe( ) As String
cs    string describe( )
java  String describe( )
py    describe( )
php   function describe( )
es    async describe( )
```

More precisely, `TYPE` is the type of the function, `NAME` it the name used for the first access to the function, `SERIAL` is the serial number of the module if the module is connected or `"unresolved"`, and `FUNCTIONID` is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1` if the module is already connected or `Relay(BadCustomeName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :
a string that describes the data logger (ex: `Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1`)

datalogger→forgetAllDataStreams() YDataLogger

Clears the data logger memory and discards all recorded data streams.

js	function **forgetAllDataStreams**( )
cpp	int **forgetAllDataStreams**( )
m	-(int) **forgetAllDataStreams**
pas	LongInt **forgetAllDataStreams**( ): LongInt
vb	function **forgetAllDataStreams**( ) As Integer
cs	int **forgetAllDataStreams**( )
java	int **forgetAllDataStreams**( )
uwp	async Task<int> **forgetAllDataStreams**( )
py	**forgetAllDataStreams**( )
php	function **forgetAllDataStreams**( )
es	async **forgetAllDataStreams**( )
dnp	int **forgetAllDataStreams**( )
cp	int **forgetAllDataStreams**( )
cmd	YDataLogger **target forgetAllDataStreams**

This method also resets the current run index to zero.

Returns :
`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→get_advertisedValue() datalogger→advertisedValue()

YDataLogger

Returns the current value of the data logger (no more than 6 characters).

```
js    function get_advertisedValue( )
cpp   string get_advertisedValue( )
m     -(NSString*) advertisedValue
pas   string get_advertisedValue( ): string
vb    function get_advertisedValue( ) As String
cs    string get_advertisedValue( )
java  String get_advertisedValue( )
uwp   async Task<string> get_advertisedValue( )
py    get_advertisedValue( )
php   function get_advertisedValue( )
es    async get_advertisedValue( )
dnp   string get_advertisedValue( )
cp    string get_advertisedValue( )
cmd   YDataLogger target get_advertisedValue
```

Returns :

a string corresponding to the current value of the data logger (no more than 6 characters).

On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

datalogger→get_autoStart() YDataLogger
datalogger→autoStart()

Returns the default activation state of the data logger on power up.

js	function **get_autoStart**( )
cpp	Y_AUTOSTART_enum **get_autoStart**( )
m	-(Y_AUTOSTART_enum) autoStart
pas	Integer **get_autoStart**( ): Integer
vb	function **get_autoStart**( ) As Integer
cs	int **get_autoStart**( )
java	int **get_autoStart**( )
uwp	async Task<int> **get_autoStart**( )
py	**get_autoStart**( )
php	function **get_autoStart**( )
es	async **get_autoStart**( )
dnp	int **get_autoStart**( )
cp	int **get_autoStart**( )
cmd	YDataLogger **target get_autoStart**

Returns :

either `Y_AUTOSTART_OFF` or `Y_AUTOSTART_ON`, according to the default activation state of the data logger on power up

On failure, throws an exception or returns `Y_AUTOSTART_INVALID`.

datalogger→get_beaconDriven() datalogger→beaconDriven()

YDataLogger

Returns true if the data logger is synchronised with the localization beacon.

```
js      function get_beaconDriven( )
cpp     Y_BEACONDRIVEN_enum get_beaconDriven( )
m       -(Y_BEACONDRIVEN_enum) beaconDriven
pas     Integer get_beaconDriven( ): Integer
vb      function get_beaconDriven( ) As Integer
cs      int get_beaconDriven( )
java    int get_beaconDriven( )
uwp     async Task<int> get_beaconDriven( )
py      get_beaconDriven( )
php     function get_beaconDriven( )
es      async get_beaconDriven( )
dnp     int get_beaconDriven( )
cp      int get_beaconDriven( )
cmd     YDataLogger target get_beaconDriven
```

Returns :

either `Y_BEACONDRIVEN_OFF` or `Y_BEACONDRIVEN_ON`, according to true if the data logger is synchronised with the localization beacon

On failure, throws an exception or returns `Y_BEACONDRIVEN_INVALID`.

datalogger→get_currentRunIndex() YDataLogger
datalogger→currentRunIndex()

Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.

js	function **get_currentRunIndex**( )
cpp	int **get_currentRunIndex**( )
m	-(int) currentRunIndex
pas	LongInt **get_currentRunIndex**( ): LongInt
vb	function **get_currentRunIndex**( ) As Integer
cs	int **get_currentRunIndex**( )
java	int **get_currentRunIndex**( )
uwp	async Task<int> **get_currentRunIndex**( )
py	**get_currentRunIndex**( )
php	function **get_currentRunIndex**( )
es	async **get_currentRunIndex**( )
dnp	int **get_currentRunIndex**( )
cp	int **get_currentRunIndex**( )
cmd	YDataLogger **target get_currentRunIndex**

Returns :

an integer corresponding to the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point

On failure, throws an exception or returns `Y_CURRENTRUNINDEX_INVALID`.

datalogger→get_dataSets() datalogger→dataSets()

YDataLogger

Returns a list of `YDataSet` objects that can be used to retrieve all measures stored by the data logger.

js	function **get_dataSets**( )
cpp	vector<YDataSet> **get_dataSets**( )
m	-(NSMutableArray*) dataSets
pas	TYDataSetArray **get_dataSets**( ): TYDataSetArray
vb	function **get_dataSets**( ) As List
cs	List<YDataSet> **get_dataSets**( )
java	ArrayList<YDataSet> **get_dataSets**( )
uwp	async Task<List<YDataSet>> **get_dataSets**( )
py	**get_dataSets**( )
php	function **get_dataSets**( )
es	async **get_dataSets**( )
dnp	YDataSetProxy[] **get_dataSets**( )
cmd	YDataLogger **target get_dataSets**

This function only works if the device uses a recent firmware, as `YDataSet` objects are not supported by firmwares older than version 13000.

Returns :
a list of `YDataSet` object.

On failure, throws an exception or returns an empty list.

datalogger→get_dataStreams() YDataLogger
datalogger→dataStreams()

Builds a list of all data streams hold by the data logger (legacy method).

The caller must pass by reference an empty array to hold `DataStream` objects, and the function fills it with objects describing available data sequences.

This is the old way to retrieve data from the DataLogger. For new applications, you should rather use `get_dataSets()` method, or call directly `get_recordedData()` on the sensor object.

Parameters :

v an array of `DataStream` objects to be filled in

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→get_errorMessage() YDataLogger
datalogger→errorMessage()

Returns the error message of the latest error with the data logger.

```
js    function get_errorMessage( )
cpp   string get_errorMessage( )
m     -(NSString*) errorMessage
pas   string get_errorMessage( ): string
vb    function get_errorMessage( ) As String
cs    string get_errorMessage( )
java  String get_errorMessage( )
py    get_errorMessage( )
php   function get_errorMessage( )
es    get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the data logger object

datalogger→get_errorType() datalogger→errorType()

YDataLogger

Returns the numerical error code of the latest error with the data logger.

```
js    function get_errorType( )
cpp   YRETCODE get_errorType( )
m     -(YRETCODE) errorType
pas   YRETCODE get_errorType( ): YRETCODE
vb    function get_errorType( ) As YRETCODE
cs    YRETCODE get_errorType( )
java  int get_errorType( )
py    get_errorType( )
php   function get_errorType( )
es    get_errorType( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the data logger object

datalogger→get_friendlyName() datalogger→friendlyName()

YDataLogger

Returns a global identifier of the data logger in the format `MODULE_NAME.FUNCTION_NAME`.

js	function **get_friendlyName**( )
cpp	string **get_friendlyName**( )
m	-(NSString*) friendlyName
cs	string **get_friendlyName**( )
java	String **get_friendlyName**( )
py	**get_friendlyName**( )
php	function **get_friendlyName**( )
es	async **get_friendlyName**( )
dnp	string **get_friendlyName**( )
cp	string **get_friendlyName**( )

The returned string uses the logical names of the module and of the data logger if they are defined, otherwise the serial number of the module and the hardware identifier of the data logger (for example: `MyCustomName.relay1`)

Returns :
a string that uniquely identifies the data logger using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `Y_FRIENDLYNAME_INVALID`.

datalogger→get_functionDescriptor() datalogger→functionDescriptor() YDataLogger

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

js	function **get_functionDescriptor**( )
cpp	YFUN_DESCR **get_functionDescriptor**( )
m	-(YFUN_DESCR) functionDescriptor
pas	YFUN_DESCR **get_functionDescriptor**( ): YFUN_DESCR
vb	function **get_functionDescriptor**( ) As YFUN_DESCR
cs	YFUN_DESCR **get_functionDescriptor**( )
java	String **get_functionDescriptor**( )
py	**get_functionDescriptor**( )
php	function **get_functionDescriptor**( )
es	async **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :
an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

datalogger→get_functionId() datalogger→functionId()

YDataLogger

Returns the hardware identifier of the data logger, without reference to the module.

js	function **get_functionId**( )
cpp	string **get_functionId**( )
m	-(NSString*) functionId
vb	function **get_functionId**( ) As String
cs	string **get_functionId**( )
java	String **get_functionId**( )
py	**get_functionId**( )
php	function **get_functionId**( )
es	async **get_functionId**( )
dnp	string **get_functionId**( )
cp	string **get_functionId**( )

For example `relay1`

Returns :
a string that identifies the data logger (ex: `relay1`)

On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

datalogger→get_hardwareId() YDataLogger
datalogger→hardwareId()

Returns the unique hardware identifier of the data logger in the form `SERIAL.FUNCTIONID`.

js	function **get_hardwareId**( )
cpp	string **get_hardwareId**( )
m	-(NSString*) hardwareId
vb	function **get_hardwareId**( ) As String
cs	string **get_hardwareId**( )
java	String **get_hardwareId**( )
py	**get_hardwareId**( )
php	function **get_hardwareId**( )
es	async **get_hardwareId**( )
dnp	string **get_hardwareId**( )
cp	string **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the data logger (for example `RELAYLO1-123456.relay1`).

Returns :
a string that uniquely identifies the data logger (ex: `RELAYLO1-123456.relay1`)

On failure, throws an exception or returns `Y_HARDWAREID_INVALID`.

datalogger→get_logicalName() datalogger→logicalName()

YDataLogger

Returns the logical name of the data logger.

js	function **get_logicalName**( )
cpp	string **get_logicalName**( )
m	-(NSString*) logicalName
pas	string **get_logicalName**( ): string
vb	function **get_logicalName**( ) As String
cs	string **get_logicalName**( )
java	String **get_logicalName**( )
uwp	async Task<string> **get_logicalName**( )
py	**get_logicalName**( )
php	function **get_logicalName**( )
es	async **get_logicalName**( )
dnp	string **get_logicalName**( )
cp	string **get_logicalName**( )
cmd	YDataLogger **target get_logicalName**

Returns :

a string corresponding to the logical name of the data logger.

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

datalogger→get_module() datalogger→module() YDataLogger

Gets the `YModule` object for the device on which the function is located.

js	function **get_module**( )
cpp	YModule * **get_module**( )
m	-(YModule*) module
pas	TYModule **get_module**( ): TYModule
vb	function **get_module**( ) As YModule
cs	YModule **get_module**( )
java	YModule **get_module**( )
py	**get_module**( )
php	function **get_module**( )
es	async **get_module**( )
dnp	YModuleProxy **get_module**( )
cp	YModuleProxy * **get_module**( )

If the function cannot be located on any module, the returned instance of `YModule` is not shown as online.

Returns :
an instance of `YModule`

datalogger→get_module_async() datalogger→module_async() YDataLogger

Gets the `YModule` object for the device on which the function is located (asynchronous version).

```
js function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned `YModule` object does not show as online.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking Firefox JavaScript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous JavasSript calls for more details.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested `YModule` object

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

datalogger→get_recording() datalogger→recording()

YDataLogger

Returns the current activation state of the data logger.

```
js      function get_recording( )
cpp     Y_RECORDING_enum get_recording( )
m       -(Y_RECORDING_enum) recording
pas     Integer get_recording( ): Integer
vb      function get_recording( ) As Integer
cs      int get_recording( )
java    int get_recording( )
uwp     async Task<int> get_recording( )
py      get_recording( )
php     function get_recording( )
es      async get_recording( )
dnp     int get_recording( )
cp      int get_recording( )
cmd     YDataLogger target get_recording
```

Returns :

a value among `Y_RECORDING_OFF`, `Y_RECORDING_ON` and `Y_RECORDING_PENDING` corresponding to the current activation state of the data logger

On failure, throws an exception or returns `Y_RECORDING_INVALID`.

datalogger→get_serialNumber() YDataLogger
datalogger→serialNumber()

Returns the serial number of the module, as set by the factory.

js	function **get_serialNumber**( )
cpp	string **get_serialNumber**( )
m	-(NSString*) serialNumber
pas	string **get_serialNumber**( ): string
vb	function **get_serialNumber**( ) As String
cs	string **get_serialNumber**( )
java	String **get_serialNumber**( )
uwp	async Task<string> **get_serialNumber**( )
py	**get_serialNumber**( )
php	function **get_serialNumber**( )
es	async **get_serialNumber**( )
dnp	string **get_serialNumber**( )
cp	string **get_serialNumber**( )
cmd	YDataLogger **target get_serialNumber**

Returns :
a string corresponding to the serial number of the module, as set by the factory.

On failure, throws an exception or returns YModule.SERIALNUMBER_INVALID.

datalogger→get_timeUTC() datalogger→timeUTC()

YDataLogger

Returns the Unix timestamp for current UTC time, if known.

```
js      function get_timeUTC( )
cpp     s64 get_timeUTC( )
m       -(s64) timeUTC
pas     int64 get_timeUTC( ): int64
vb      function get_timeUTC( ) As Long
cs      long get_timeUTC( )
java    long get_timeUTC( )
uwp     async Task<long> get_timeUTC( )
py      get_timeUTC( )
php     function get_timeUTC( )
es      async get_timeUTC( )
dnp     long get_timeUTC( )
cp      s64 get_timeUTC( )
cmd     YDataLogger target get_timeUTC
```

Returns :

an integer corresponding to the Unix timestamp for current UTC time, if known

On failure, throws an exception or returns `Y_TIMEUTC_INVALID`.

datalogger→get_usage() datalogger→usage()

YDataLogger

Returns the percentage of datalogger memory in use.

```
js    function get_usage( )
cpp   int get_usage( )
m     -(int) usage
pas   LongInt get_usage( ): LongInt
vb    function get_usage( ) As Integer
cs    int get_usage( )
java  int get_usage( )
uwp   async Task<int> get_usage( )
py    get_usage( )
php   function get_usage( )
es    async get_usage( )
dnp   int get_usage( )
cp    int get_usage( )
cmd   YDataLogger target get_usage
```

Returns :

an integer corresponding to the percentage of datalogger memory in use

On failure, throws an exception or returns `Y_USAGE_INVALID`.

datalogger→get_userData() datalogger→userData()

YDataLogger

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
js      function get_userData( )
cpp     void * get_userData( )
m       -(id) userData
pas     Tobject get_userData( ): Tobject
vb      function get_userData( ) As Object
cs      object get_userData( )
java    Object get_userData( )
py      get_userData( )
php     function get_userData( )
es      async get_userData( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

datalogger→isOnline() YDataLogger

Checks if the data logger is currently reachable, without raising any error.

js	function **isOnline**( )
cpp	bool **isOnline**( )
m	-(BOOL) **isOnline**
pas	boolean **isOnline**( ): boolean
vb	function **isOnline**( ) As Boolean
cs	bool **isOnline**( )
java	boolean **isOnline**( )
py	**isOnline**( )
php	function **isOnline**( )
es	async **isOnline**( )
dnp	bool **isOnline**( )
cp	bool **isOnline**( )

If there is a cached value for the data logger in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the data logger.

Returns :
`true` if the data logger can be reached, and `false` otherwise

datalogger→isOnline_async() YDataLogger

Checks if the data logger is currently reachable, without raising any error (asynchronous version).

```
js function isOnline_async( callback, context)
```

If there is a cached value for the data logger in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

datalogger→isReadOnly() YDataLogger

Test if the function is readOnly.

cpp	bool **isReadOnly**( )
m	-(bool) **isReadOnly**
pas	boolean **isReadOnly**( ): boolean
vb	function **isReadOnly**( ) As Boolean
cs	bool **isReadOnly**( )
java	boolean **isReadOnly**( )
uwp	async Task<bool> **isReadOnly**( )
py	**isReadOnly**( )
php	function **isReadOnly**( )
es	async **isReadOnly**( )
dnp	bool **isReadOnly**( )
cp	bool **isReadOnly**( )
cmd	YDataLogger **target isReadOnly**

Return `true` if the function is write protected or that the function is not available.

Returns :
`true` if the function is readOnly or not online.

datalogger→load() YDataLogger

Preloads the data logger cache with a specified validity duration.

```
js    function load( msValidity)
cpp   YRETCODE load( int msValidity)
m     -(YRETCODE) load : (u64) msValidity
pas   YRETCODE load( msValidity: u64): YRETCODE
vb    function load( ByVal msValidity As Long) As YRETCODE
cs    YRETCODE load( ulong msValidity)
java  int load( long msValidity)
py    load( msValidity)
php   function load( $msValidity)
es    async load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→loadAttribute() YDataLogger

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
js    function loadAttribute( attrName)
cpp   string loadAttribute( string attrName)
m     -(NSString*) loadAttribute : (NSString*) attrName
pas   string loadAttribute( attrName: string): string
vb    function loadAttribute( ByVal attrName As String) As String
cs    string loadAttribute( string attrName)
java  String loadAttribute( String attrName)
uwp   async Task<string> loadAttribute( string attrName)
py    loadAttribute( attrName)
php   function loadAttribute( $attrName)
es    async loadAttribute( attrName)
dnp   string loadAttribute( string attrName)
cp    string loadAttribute( string attrName)
```

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

datalogger→load_async() YDataLogger

Preloads the data logger cache with a specified validity duration (asynchronous version).

```
js function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking the JavaScript virtual machine.

Parameters :

msValidity an integer corresponding to the validity of the loaded function parameters, in milliseconds

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI_SUCCESS`)

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

datalogger→muteValueCallbacks() YDataLogger

Disables the propagation of every new advertised value to the parent hub.

js	function **muteValueCallbacks**( )
cpp	int **muteValueCallbacks**( )
m	-(int) **muteValueCallbacks**
pas	LongInt **muteValueCallbacks**( ): LongInt
vb	function **muteValueCallbacks**( ) As Integer
cs	int **muteValueCallbacks**( )
java	int **muteValueCallbacks**( )
uwp	async Task<int> **muteValueCallbacks**( )
py	**muteValueCallbacks**( )
php	function **muteValueCallbacks**( )
es	async **muteValueCallbacks**( )
dnp	int **muteValueCallbacks**( )
cp	int **muteValueCallbacks**( )
cmd	YDataLogger **target muteValueCallbacks**

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→nextDataLogger() YDataLogger

Continues the enumeration of data loggers started using `yFirstDataLogger()`.

js	function **nextDataLogger**( )
cpp	YDataLogger * **nextDataLogger**( )
m	-(YDataLogger*) **nextDataLogger**
pas	TYDataLogger **nextDataLogger**( ): TYDataLogger
vb	function **nextDataLogger**( ) As YDataLogger
cs	YDataLogger **nextDataLogger**( )
java	YDataLogger **nextDataLogger**( )
uwp	YDataLogger **nextDataLogger**( )
py	**nextDataLogger**( )
php	function **nextDataLogger**( )
es	**nextDataLogger**( )

Caution: You can't make any assumption about the returned data loggers order. If you want to find a specific a data logger, use `DataLogger.findDataLogger()` and a hardwareID or a logical name.

Returns :

a pointer to a `YDataLogger` object, corresponding to a data logger currently online, or a `null` pointer if there are no more data loggers to enumerate.

datalogger→registerValueCallback() YDataLogger

Registers the callback function that is invoked on every change of advertised value.

js	function **registerValueCallback**(**callback**)
cpp	int **registerValueCallback**(YDataLoggerValueCallback **callback**)
m	-(int) **registerValueCallback** : (YDataLoggerValueCallback) **callback**
pas	LongInt **registerValueCallback**(**callback**: TYDataLoggerValueCallback): LongInt
vb	function **registerValueCallback**(ByVal **callback** As YDataLoggerValueCallback) As Integer
cs	int **registerValueCallback**(ValueCallback **callback**)
java	int **registerValueCallback**(UpdateCallback **callback**)
uwp	async Task<int> **registerValueCallback**(ValueCallback **callback**)
py	**registerValueCallback**(**callback**)
php	function **registerValueCallback**(**$callback**)
es	async **registerValueCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

datalogger→set_autoStart() datalogger→setAutoStart() YDataLogger

Changes the default activation state of the data logger on power up.

```
js    function set_autoStart( newval)
cpp   int set_autoStart( Y_AUTOSTART_enum newval)
m     -(int) setAutoStart : (Y_AUTOSTART_enum) newval
pas   integer set_autoStart( newval: Integer): integer
vb    function set_autoStart( ByVal newval As Integer) As Integer
cs    int set_autoStart( int newval)
java  int set_autoStart( int newval)
uwp   async Task<int> set_autoStart( int newval)
py    set_autoStart( newval)
php   function set_autoStart( $newval)
es    async set_autoStart( newval)
dnp   int set_autoStart( int newval)
cp    int set_autoStart( int newval)
cmd   YDataLogger target set_autoStart newval
```

Do not forget to call the `saveToFlash()` method of the module to save the configuration change. Note: if the device doesn't have any time source at his disposal when starting up, it will wait for ~8 seconds before automatically starting to record with an arbitrary timestamp

Parameters :

newval either `Y_AUTOSTART_OFF` or `Y_AUTOSTART_ON`, according to the default activation state of the data logger on power up

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→set_beaconDriven() datalogger→setBeaconDriven() YDataLogger

Changes the type of synchronisation of the data logger.

```
js    function set_beaconDriven( newval)
cpp   int set_beaconDriven( Y_BEACONDRIVEN_enum newval)
m     -(int) setBeaconDriven : (Y_BEACONDRIVEN_enum) newval
pas   integer set_beaconDriven( newval: Integer): integer
vb    function set_beaconDriven( ByVal newval As Integer) As Integer
cs    int set_beaconDriven( int newval)
java  int set_beaconDriven( int newval)
uwp   async Task<int> set_beaconDriven( int newval)
py    set_beaconDriven( newval)
php   function set_beaconDriven( $newval)
es    async set_beaconDriven( newval)
dnp   int set_beaconDriven( int newval)
cp    int set_beaconDriven( int newval)
cmd   YDataLogger target set_beaconDriven newval
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval either `Y_BEACONDRIVEN_OFF` or `Y_BEACONDRIVEN_ON`, according to the type of synchronisation of the data logger

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→set_logicalName() datalogger→setLogicalName() YDataLogger

Changes the logical name of the data logger.

```
js    function set_logicalName( newval)
cpp   int set_logicalName( string newval)
m     -(int) setLogicalName : (NSString*) newval
pas   integer set_logicalName( newval: string): integer
vb    function set_logicalName( ByVal newval As String) As Integer
cs    int set_logicalName( string newval)
java  int set_logicalName( String newval)
uwp   async Task<int> set_logicalName( string newval)
py    set_logicalName( newval)
php   function set_logicalName( $newval)
es    async set_logicalName( newval)
dnp   int set_logicalName( string newval)
cp    int set_logicalName( string newval)
cmd   YDataLogger target set_logicalName newval
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the data logger.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→set_recording() datalogger→setRecording()

YDataLogger

Changes the activation state of the data logger to start/stop recording data.

```
js    function set_recording( newval)
cpp   int set_recording( Y_RECORDING_enum newval)
m     -(int) setRecording : (Y_RECORDING_enum) newval
pas   integer set_recording( newval: Integer): integer
vb    function set_recording( ByVal newval As Integer) As Integer
cs    int set_recording( int newval)
java  int set_recording( int newval)
uwp   async Task<int> set_recording( int newval)
py    set_recording( newval)
php   function set_recording( $newval)
es    async set_recording( newval)
dnp   int set_recording( int newval)
cp    int set_recording( int newval)
cmd   YDataLogger target set_recording newval
```

Parameters :

newval a value among `Y_RECORDING_OFF`, `Y_RECORDING_ON` and `Y_RECORDING_PENDING` corresponding to the activation state of the data logger to start/stop recording data

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→set_timeUTC() datalogger→setTimeUTC() YDataLogger

Changes the current UTC time reference used for recorded data.

```
js    function set_timeUTC( newval)
cpp   int set_timeUTC( s64 newval)
m     -(int) setTimeUTC : (s64) newval
pas   integer set_timeUTC( newval: int64): integer
vb    function set_timeUTC( ByVal newval As Long) As Integer
cs    int set_timeUTC( long newval)
java  int set_timeUTC( long newval)
uwp   async Task<int> set_timeUTC( long newval)
py    set_timeUTC( newval)
php   function set_timeUTC( $newval)
es    async set_timeUTC( newval)
dnp   int set_timeUTC( long newval)
cp    int set_timeUTC( s64 newval)
cmd   YDataLogger target set_timeUTC newval
```

Parameters :

newval an integer corresponding to the current UTC time reference used for recorded data

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→set_userData() datalogger→setUserData() YDataLogger

Stores a user context provided as argument in the userData attribute of the function.

```
js    function set_userData( data)
cpp   void set_userData( void * data)
m     -(void) setUserData : (id) data
pas   set_userData( data: Tobject)
vb    procedure set_userData( ByVal data As Object)
cs    void set_userData( object data)
java  void set_userData( Object data)
py    set_userData( data)
php   function set_userData( $data)
es    async set_userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

datalogger→unmuteValueCallbacks() YDataLogger

Re-enables the propagation of every new advertised value to the parent hub.

js	function **unmuteValueCallbacks**( )
cpp	int **unmuteValueCallbacks**( )
m	-(int) **unmuteValueCallbacks**
pas	LongInt **unmuteValueCallbacks**( ): LongInt
vb	function **unmuteValueCallbacks**( ) As Integer
cs	int **unmuteValueCallbacks**( )
java	int **unmuteValueCallbacks**( )
uwp	async Task<int> **unmuteValueCallbacks**( )
py	**unmuteValueCallbacks**( )
php	function **unmuteValueCallbacks**( )
es	async **unmuteValueCallbacks**( )
dnp	int **unmuteValueCallbacks**( )
cp	int **unmuteValueCallbacks**( )
cmd	YDataLogger **target unmuteValueCallbacks**

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI_SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→wait_async() YDataLogger

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js  function wait_async( callback, context)
es  wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

24.7. Class YDataSet

Recorded data sequence, as returned by `sensor.get_recordedData()`

`YDataSet` objects make it possible to retrieve a set of recorded measures for a given sensor and a specified time interval. They can be used to load data points with a progress report. When the `YDataSet` object is instantiated by the `sensor.get_recordedData()` function, no data is yet loaded from the module. It is only when the `loadMore()` method is called over and over than data will be effectively loaded from the dataLogger.

A preview of available measures is available using the function `get_preview()` as soon as `loadMore()` has been called once. Measures themselves are available using function `get_measures()` when loaded by subsequent calls to `loadMore()`.

This class can only be used on devices that use a relatively recent firmware, as `YDataSet` objects are not supported by firmwares older than version 13000.

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_module.js'></script>
cpp	#include "yocto_module.h"
m	#import "yocto_module.h"
pas	uses yocto_module;
vb	yocto_module.vb
cs	yocto_module.cs
java	import com.yoctopuce.YoctoAPI.YDataSet;
uwp	import com.yoctopuce.YoctoAPI.YDataSet;
py	from yocto_module import *
php	require_once('yocto_module.php');
es	in HTML: <script src="../../lib/yocto_module.js"></script> in node.js: require('yoctolib-es2017/yocto_module.js');
dnp	import YoctoProxyAPI.YDataSetProxy
cp	#include "yocto_module_proxy.h"
ml	import YoctoProxyAPI.YDataSetProxy

Global functions
YDataSet.Init(**sensorName**, **startTime**, **endTime**) Retrieves a `YDataSet` object holding historical data for a sensor given by its name or hardware identifier, for a specified time interval.

`YDataSet` methods
dataset→get_endTimeUTC() Returns the end time of the dataset, relative to the Jan 1, 1970.
dataset→get_functionId() Returns the hardware identifier of the function that performed the measure, without reference to the module.
dataset→get_hardwareId() Returns the unique hardware identifier of the function who performed the measures, in the form `SERIAL.FUNCTIONID`.
dataset→get_measures() Returns all measured values currently available for this DataSet, as a list of `YMeasure` objects.
dataset→get_measuresAt(**measure**)

Returns the detailed set of measures for the time interval corresponding to a given condensed measures previously returned by `get_preview()`.

dataset→get_measuresAvgAt(index)

Returns the average value observed during the time interval covered by the specified entry in the preview.

dataset→get_measuresEndTimeAt(index)

Returns the end time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

dataset→get_measuresMaxAt(index)

Returns the largest value observed during the time interval covered by the specified entry in the preview.

dataset→get_measuresMinAt(index)

Returns the smallest value observed during the time interval covered by the specified entry in the preview.

dataset→get_measuresRecordCount()

Returns the number of measurements currently loaded for this data set.

dataset→get_measuresStartTimeAt(index)

Returns the start time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

dataset→get_preview()

Returns a condensed version of the measures that can retrieved in this `YDataSet`, as a list of `YMeasure` objects.

dataset→get_previewAvgAt(index)

Returns the average value observed during the time interval covered by the specified entry in the preview.

dataset→get_previewEndTimeAt(index)

Returns the end time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

dataset→get_previewMaxAt(index)

Returns the largest value observed during the time interval covered by the specified entry in the preview.

dataset→get_previewMinAt(index)

Returns the smallest value observed during the time interval covered by the specified entry in the preview.

dataset→get_previewRecordCount()

Returns the number of entries in the preview summarizing this data set

dataset→get_previewStartTimeAt(index)

Returns the start time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

dataset→get_progress()

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

dataset→get_startTimeUTC()

Returns the start time of the dataset, relative to the Jan 1, 1970.

dataset→get_summary()

Returns an `YMeasure` object which summarizes the whole `YDataSet`.

dataset→get_summaryAvg()

Returns the average value observed during the time interval covered by this data set.

dataset→get_summaryEndTime()

Returns the end time of the last measure in the data set, relative to the Jan 1, 1970 UTC (Unix timestamp).

dataset→get_summaryMax()

Returns the largest value observed during the time interval covered by this data set.

dataset→get_summaryMin()

Returns the smallest value observed during the time interval covered by this data set.

dataset→get_summaryStartTime()

Returns the start time of the first measure in the data set, relative to the Jan 1, 1970 UTC (Unix timestamp).

dataset→get_unit()

Returns the measuring unit for the measured value.

dataset→loadMore()

Loads the the next block of measures from the dataLogger, and updates the progress indicator.

dataset→loadMore_async(callback, context)

Loads the the next block of measures from the dataLogger asynchronously.

YDataSet.Init() YDataSet

YDataSet.Init()

Retrieves a `YDataSet` object holding historical data for a sensor given by its name or hardware identifier, for a specified time interval.

The measures will be retrieved from the data logger, which must have been turned on at the desired time. Methods of the `YDataSet` class makes it possible to get an overview of the recorded data, and to load progressively a large set of measures from the data logger.

Parameters :

sensorName logical name or hardware identifier of the sensor for which data logger records are requested.

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without ending limit.

Returns :

an instance of `YDataSet`, providing access to historical data. Past measures can be loaded progressively using methods from the `YDataSet` object.

dataset→get_endTimeUTC() YDataSet
dataset→endTimeUTC()

Returns the end time of the dataset, relative to the Jan 1, 1970.

```
js    function get_endTimeUTC( )
cpp   s64 get_endTimeUTC( )
m     -(s64) endTimeUTC
pas   int64 get_endTimeUTC( ): int64
vb    function get_endTimeUTC( ) As Long
cs    long get_endTimeUTC( )
java  long get_endTimeUTC( )
uwp   async Task<long> get_endTimeUTC( )
py    get_endTimeUTC( )
php   function get_endTimeUTC( )
es    async get_endTimeUTC( )
dnp   long get_endTimeUTC( )
cp    s64 get_endTimeUTC( )
```

When the `YDataSet` object is created, the end time is the value passed in parameter to the `get_dataSet()` function. After the very first call to `loadMore()`, the end time is updated to reflect the timestamp of the last measure actually found in the dataLogger within the specified range.

DEPRECATED: This method has been replaced by `get_summary()` which contain more precise informations.

Returns :
an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the end of this data set (i.e. Unix time representation of the absolute time).

dataset→get_functionId() YDataSet
dataset→functionId()

Returns the hardware identifier of the function that performed the measure, without reference to the module.

js	function **get_functionId**( )
cpp	string **get_functionId**( )
m	-(NSString*) functionId
pas	string **get_functionId**( ): string
vb	function **get_functionId**( ) As String
cs	string **get_functionId**( )
java	String **get_functionId**( )
uwp	async Task<string> **get_functionId**( )
py	**get_functionId**( )
php	function **get_functionId**( )
es	async **get_functionId**( )
dnp	string **get_functionId**( )
cp	string **get_functionId**( )

For example `temperature1`.

Returns :
a string that identifies the function (ex: `temperature1`)

dataset→get_hardwareId() YDataSet
dataset→hardwareId()

Returns the unique hardware identifier of the function who performed the measures, in the form `SERIAL.FUNCTIONID`.

js	function **get_hardwareId**( )
cpp	string **get_hardwareId**( )
m	-(NSString*) hardwareId
pas	string **get_hardwareId**( ): string
vb	function **get_hardwareId**( ) As String
cs	string **get_hardwareId**( )
java	String **get_hardwareId**( )
uwp	async Task<string> **get_hardwareId**( )
py	**get_hardwareId**( )
php	function **get_hardwareId**( )
es	async **get_hardwareId**( )
dnp	string **get_hardwareId**( )
cp	string **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( )
cpp   vector<YMeasure> get_measures( )
m     -(NSMutableArray*) measures
pas   TYMeasureArray get_measures( ): TYMeasureArray
vb    function get_measures( ) As List
cs    List<YMeasure> get_measures( )
java  ArrayList<YMeasure> get_measures( )
uwp   async Task<List<YMeasure>> get_measures( )
py    get_measures( )
php   function get_measures( )
es    async get_measures( )
dnp   YMeasure[] get_measures( )
cp    vector<YMeasure> get_measures( )
```

Each item includes: - the start of the measure time interval - the end of the measure time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

Before calling this method, you should call `loadMore()` to load data from the device. You may have to call loadMore() several time until all rows are loaded, but you can start looking at available data rows before the load is complete.

The oldest measures are always loaded first, and the most recent measures will be loaded last. As a result, timestamps are normally sorted in ascending order within the measure table, unless there was an unexpected adjustment of the datalogger UTC clock.

Returns :
a table of records, where each record depicts the measured value for a given time interval

On failure, throws an exception or returns an empty array.

dataset→get_measuresAt() YDataSet
dataset→measuresAt()

Returns the detailed set of measures for the time interval corresponding to a given condensed measures previously returned by `get_preview()`.

js	function **get_measuresAt**(**measure**)
cpp	vector<YMeasure> **get_measuresAt**(YMeasure **measure**)
m	-(NSMutableArray*) measuresAt : (YMeasure*) **measure**
pas	TYMeasureArray **get_measuresAt**(**measure**: TYMeasure): TYMeasureArray
vb	function **get_measuresAt**(ByVal **measure** As YMeasure) As List
cs	List<YMeasure> **get_measuresAt**(YMeasure **measure**)
java	ArrayList<YMeasure> **get_measuresAt**(YMeasure **measure**)
uwp	async Task<List<YMeasure>> **get_measuresAt**(YMeasure **measure**)
py	**get_measuresAt**(**measure**)
php	function **get_measuresAt**(**$measure**)
es	async **get_measuresAt**(**measure**)
dnp	YMeasure[] **get_measuresAt**(YMeasure **measure**)
cp	vector<YMeasure> **get_measuresAt**(YMeasure **measure**)

The result is provided as a list of `YMeasure` objects.

Parameters :

measure condensed measure from the list previously returned by `get_preview()`.

Returns :

a table of records, where each record depicts the measured values during a time interval

On failure, throws an exception or returns an empty array.

dataset→get_measuresAvgAt() YDataSet
dataset→measuresAvgAt()

Returns the average value observed during the time interval covered by the specified entry in the preview.

Parameters :

index an integer index in the range [`0`...`MeasuresRecordCount`-1].

Returns :

a floating-point number corresponding to the average value observed.

dataset→get_measuresEndTimeAt() YDataSet
dataset→measuresEndTimeAt()

Returns the end time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

Parameters :

index an integer index in the range [`0`...`MeasuresRecordCount`-1].

Returns :

a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

dataset→get_measuresMaxAt() YDataSet
dataset→measuresMaxAt()

Returns the largest value observed during the time interval covered by the specified entry in the preview.

Parameters :

index an integer index in the range [`0`...`MeasuresRecordCount`-1].

Returns :

a floating-point number corresponding to the largest value observed.

dataset→get_measuresMinAt() YDataSet
dataset→measuresMinAt()

Returns the smallest value observed during the time interval covered by the specified entry in the preview.

Parameters :

index an integer index in the range [`0`...`MeasuresRecordCount`-1].

Returns :

a floating-point number corresponding to the smallest value observed.

dataset→get_measuresRecordCount() YDataSet
dataset→measuresRecordCount()

Returns the number of measurements currently loaded for this data set.

The total number of record is only known when the data set is fully loaded, i.e. when `loadMore()` has been invoked until the progresss indicator returns 100.

Returns :
an integer number corresponding to the number of entries loaded.

dataset→get_measuresStartTimeAt() YDataSet
dataset→measuresStartTimeAt()

Returns the start time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

Parameters :

index an integer index in the range [`0`...`MeasuresRecordCount`-1].

Returns :

a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

dataset→get_preview() YDataSet
dataset→preview()

Returns a condensed version of the measures that can retrieved in this `YDataSet`, as a list of `YMeasure` objects.

js	function **get_preview**( )
cpp	vector<YMeasure> **get_preview**( )
m	-(NSMutableArray*) preview
pas	TYMeasureArray **get_preview**( ): TYMeasureArray
vb	function **get_preview**( ) As List
cs	List<YMeasure> **get_preview**( )
java	ArrayList<YMeasure> **get_preview**( )
uwp	async Task<List<YMeasure>> **get_preview**( )
py	**get_preview**( )
php	function **get_preview**( )
es	async **get_preview**( )
dnp	YMeasure[] **get_preview**( )
cp	vector<YMeasure> **get_preview**( )

Each item includes: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This preview is available as soon as `loadMore()` has been called for the first time.

Returns :
a table of records, where each record depicts the measured values during a time interval

On failure, throws an exception or returns an empty array.

dataset→get_previewAvgAt() YDataSet
dataset→previewAvgAt()

Returns the average value observed during the time interval covered by the specified entry in the preview.

Parameters :

index an integer index in the range [`0`...`PreviewRecordCount`-1].

Returns :

a floating-point number corresponding to the average value observed.

dataset→get_previewEndTimeAt() YDataSet
dataset→previewEndTimeAt()

Returns the end time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

Parameters :

index an integer index in the range [`0`...`PreviewRecordCount`-1].

Returns :

a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

dataset→get_previewMaxAt() YDataSet
dataset→previewMaxAt()

Returns the largest value observed during the time interval covered by the specified entry in the preview.

Parameters :

index an integer index in the range [`0`...`PreviewRecordCount`-1].

Returns :

a floating-point number corresponding to the largest value observed.

dataset→get_previewMinAt() YDataSet
dataset→previewMinAt()

Returns the smallest value observed during the time interval covered by the specified entry in the preview.

Parameters :

index an integer index in the range [`0`...`PreviewRecordCount`-1].

Returns :

a floating-point number corresponding to the smallest value observed.

dataset→get_previewRecordCount() YDataSet
dataset→previewRecordCount()

Returns the number of entries in the preview summarizing this data set

Returns :
an integer number corresponding to the number of entries.

dataset→get_previewStartTimeAt() YDataSet
dataset→previewStartTimeAt()

Returns the start time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

Parameters :

index an integer index in the range [`0`...`PreviewRecordCount`-1].

Returns :

a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

dataset→get_progress() YDataSet
dataset→progress()

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

js	function **get_progress**( )
cpp	int **get_progress**( )
m	-(int) progress
pas	LongInt **get_progress**( ): LongInt
vb	function **get_progress**( ) As Integer
cs	int **get_progress**( )
java	int **get_progress**( )
uwp	async Task<int> **get_progress**( )
py	**get_progress**( )
php	function **get_progress**( )
es	async **get_progress**( )
dnp	int **get_progress**( )
cp	int **get_progress**( )

When the object is instantiated by `get_dataSet`, the progress is zero. Each time `loadMore()` is invoked, the progress is updated, to reach the value 100 only once all measures have been loaded.

Returns :
an integer in the range 0 to 100 (percentage of completion).

dataset→get_startTimeUTC() YDataSet
dataset→startTimeUTC()

Returns the start time of the dataset, relative to the Jan 1, 1970.

js	function **get_startTimeUTC**( )
cpp	s64 **get_startTimeUTC**( )
m	-(s64) startTimeUTC
pas	int64 **get_startTimeUTC**( ): int64
vb	function **get_startTimeUTC**( ) As Long
cs	long **get_startTimeUTC**( )
java	long **get_startTimeUTC**( )
uwp	async Task<long> **get_startTimeUTC**( )
py	**get_startTimeUTC**( )
php	function **get_startTimeUTC**( )
es	async **get_startTimeUTC**( )
dnp	long **get_startTimeUTC**( )
cp	s64 **get_startTimeUTC**( )

When the `YDataSet` object is created, the start time is the value passed in parameter to the `get_dataSet()` function. After the very first call to `loadMore()`, the start time is updated to reflect the timestamp of the first measure actually found in the dataLogger within the specified range.

DEPRECATED: This method has been replaced by `get_summary()` which contain more precise informations.

Returns :
an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the beginning of this data set (i.e. Unix time representation of the absolute time).

dataset→get_summary() YDataSet
dataset→summary()

Returns an `YMeasure` object which summarizes the whole `YDataSet`.

js	function **get_summary**( )
cpp	YMeasure **get_summary**( )
m	-(YMeasure*) summary
pas	TYMeasure **get_summary**( ): TYMeasure
vb	function **get_summary**( ) As YMeasure
cs	YMeasure **get_summary**( )
java	YMeasure **get_summary**( )
uwp	async Task<YMeasure> **get_summary**( )
py	**get_summary**( )
php	function **get_summary**( )
es	async **get_summary**( )
dnp	YMeasure **get_summary**( )
cp	YMeasure **get_summary**( )

In includes the following information: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This summary is available as soon as `loadMore()` has been called for the first time.

Returns :
an `YMeasure` object

dataset→get_summaryAvg() YDataSet
dataset→summaryAvg()

Returns the average value observed during the time interval covered by this data set.

Returns :
a floating-point number corresponding to the average value observed.

dataset→get_summaryEndTime() YDataSet
dataset→summaryEndTime()

Returns the end time of the last measure in the data set, relative to the Jan 1, 1970 UTC (Unix timestamp).

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

Returns :
a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

dataset→get_summaryMax() YDataSet
dataset→summaryMax()

Returns the largest value observed during the time interval covered by this data set.

Returns :
a floating-point number corresponding to the largest value observed.

dataset→get_summaryMin() YDataSet
dataset→summaryMin()

Returns the smallest value observed during the time interval covered by this data set.

Returns :
a floating-point number corresponding to the smallest value observed.

dataset→get_summaryStartTime() YDataSet
dataset→summaryStartTime()

Returns the start time of the first measure in the data set, relative to the Jan 1, 1970 UTC (Unix timestamp).

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

Returns :
a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

dataset→get_unit() YDataSet
dataset→unit()

Returns the measuring unit for the measured value.

```
js    function get_unit( )
cpp   string get_unit( )
m     -(NSString*) unit
pas   string get_unit( ): string
vb    function get_unit( ) As String
cs    string get_unit( )
java  String get_unit( )
uwp   async Task<string> get_unit( )
py    get_unit( )
php   function get_unit( )
es    async get_unit( )
dnp   string get_unit( )
cp    string get_unit( )
```

Returns :
a string that represents a physical unit.

On failure, throws an exception or returns `Y_UNIT_INVALID`.

dataset→loadMore() YDataSet

Loads the the next block of measures from the dataLogger, and updates the progress indicator.

js	function **loadMore**( )
cpp	int **loadMore**( )
m	-(int) **loadMore**
pas	LongInt **loadMore**( ): LongInt
vb	function **loadMore**( ) As Integer
cs	int **loadMore**( )
java	int **loadMore**( )
uwp	async Task<int> **loadMore**( )
py	**loadMore**( )
php	function **loadMore**( )
es	async **loadMore**( )
dnp	int **loadMore**( )
cp	int **loadMore**( )

Returns :
an integer in the range 0 to 100 (percentage of completion), or a negative error code in case of failure.

On failure, throws an exception or returns a negative error code.

dataset→loadMore_async() YDataSet

Loads the the next block of measures from the dataLogger asynchronously.

```
js function loadMore_async( callback, context)
```

Parameters :

callback callback function that is invoked when the w The callback function receives three arguments: - the user-specific context object - the `YDataSet` object whose `loadMore_async` was invoked - the load result: either the progress indicator (0...100), or a negative error code in case of failure.

context user-specific object that is passed as-is to the callback function

Returns :

nothing.

24.8. Class YMeasure

Measured value, returned in particular by the methods of the `YDataSet` class.

`YMeasure` objects are used within the API to represent a value measured at a specified time. These objects are used in particular in conjunction with the `YDataSet` class, but also for sensors periodic timed reports (see `sensor.registerTimedReportCallback`).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_module.js'></script>
cpp	#include "yocto_module.h"
m	#import "yocto_module.h"
pas	uses yocto_module;
vb	yocto_module.vb
cs	yocto_module.cs
java	import com.yoctopuce.YoctoAPI.YMeasure;
uwp	import com.yoctopuce.YoctoAPI.YMeasure;
py	from yocto_module import *
php	require_once('yocto_module.php');
es	in HTML: <script src="../../lib/yocto_module.js"></script> in node.js: require('yoctolib-es2017/yocto_module.js');

`YMeasure` methods
measure→get_averageValue() Returns the average value observed during the time interval covered by this measure.
measure→get_endTimeUTC() Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).
measure→get_maxValue() Returns the largest value observed during the time interval covered by this measure.
measure→get_minValue() Returns the smallest value observed during the time interval covered by this measure.
measure→get_startTimeUTC() Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

measure→get_averageValue() YMeasure
measure→averageValue()

Returns the average value observed during the time interval covered by this measure.

js	function **get_averageValue**( )
cpp	double **get_averageValue**( )
m	-(double) averageValue
pas	double **get_averageValue**( ): double
vb	function **get_averageValue**( ) As Double
cs	double **get_averageValue**( )
java	double **get_averageValue**( )
uwp	double **get_averageValue**( )
py	**get_averageValue**( )
php	function **get_averageValue**( )
es	**get_averageValue**( )

Returns :
a floating-point number corresponding to the average value observed.

measure→get_endTimeUTC() YMeasure
measure→endTimeUTC()

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

js	function **get_endTimeUTC**( )
cpp	double **get_endTimeUTC**( )
m	-(double) endTimeUTC
pas	double **get_endTimeUTC**( ): double
vb	function **get_endTimeUTC**( ) As Double
cs	double **get_endTimeUTC**( )
java	double **get_endTimeUTC**( )
uwp	double **get_endTimeUTC**( )
py	**get_endTimeUTC**( )
php	function **get_endTimeUTC**( )
es	**get_endTimeUTC**( )

When the recording rate is higher than 1 sample per second, the timestamp may have a fractional part.

Returns :
a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the end of this measure.

measure→get_maxValue() YMeasure
measure→maxValue()

Returns the largest value observed during the time interval covered by this measure.

js	function **get_maxValue**( )
cpp	double **get_maxValue**( )
m	-(double) maxValue
pas	double **get_maxValue**( ): double
vb	function **get_maxValue**( ) As Double
cs	double **get_maxValue**( )
java	double **get_maxValue**( )
uwp	double **get_maxValue**( )
py	**get_maxValue**( )
php	function **get_maxValue**( )
es	**get_maxValue**( )

Returns :
a floating-point number corresponding to the largest value observed.

measure→get_minValue() YMeasure
measure→minValue()

Returns the smallest value observed during the time interval covered by this measure.

js	function **get_minValue**( )
cpp	double **get_minValue**( )
m	-(double) minValue
pas	double **get_minValue**( ): double
vb	function **get_minValue**( ) As Double
cs	double **get_minValue**( )
java	double **get_minValue**( )
uwp	double **get_minValue**( )
py	**get_minValue**( )
php	function **get_minValue**( )
es	**get_minValue**( )

Returns :
a floating-point number corresponding to the smallest value observed.

measure→get_startTimeUTC() YMeasure
measure→startTimeUTC()

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

```
js    function get_startTimeUTC( )
cpp   double get_startTimeUTC( )
m     -(double) startTimeUTC
pas   double get_startTimeUTC( ): double
vb    function get_startTimeUTC( ) As Double
cs    double get_startTimeUTC( )
java  double get_startTimeUTC( )
uwp   double get_startTimeUTC( )
py    get_startTimeUTC( )
php   function get_startTimeUTC( )
es    get_startTimeUTC( )
```

When the recording rate is higher then 1 sample per second, the timestamp may have a fractional part.

Returns :
a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

25. Troubleshooting

25.1. Where to start?

If it is the first time that you use a Yoctopuce module and you do not really know where to start, have a look at the Yoctopuce blog. There is a section dedicated to beginners [1].

25.2. Programming examples don't seem to work

Most of Yoctopuce API programming examples are command line programs and require some parameters to work properly. You have to start them from your operating system command prompt, or configure your IDE to run them with the proper parameters. [2].

25.3. Linux and USB

To work correctly under Linux, the the library needs to have write access to all the Yoctopuce USB peripherals. However, by default under Linux, USB privileges of the non-root users are limited to read access. To avoid having to run the *VirtualHub* as root, you need to create a new *udev* rule to authorize one or several users to have write access to the Yoctopuce peripherals.

To add a new *udev* rule to your installation, you must add a file with a name following the "`##-arbitraryName.rules`" format, in the "`/etc/udev/rules.d`" directory. When the system is starting, *udev* reads all the files with a "`.rules`" extension in this directory, respecting the alphabetical order (for example, the "`51-custom.rules`" file is interpreted AFTER the "`50-udev-default.rules`" file).

The "`50-udev-default`" file contains the system default *udev* rules. To modify the default behavior, you therefore need to create a file with a name that starts with a number larger than 50, that will override the system default rules. Note that to add a rule, you need a root access on the system.

In the `udev_conf` directory of the *VirtualHub* for Linux[3] archive, there are two rule examples which you can use as a basis.

[1] see: http://www.yoctopuce.com/EN/blog_by_categories/for-the-beginners
[2] see: http://www.yoctopuce.com/EN/article/about-programming-examples
[3] http://www.yoctopuce.com/FR/virtualhub.php

Example 1: 51-yoctopuce.rules

This rule provides all the users with read and write access to the Yoctopuce USB peripherals. Access rights for all other peripherals are not modified. If this scenario suits you, you only need to copy the "`51-yoctopuce_all.rules`" file into the "`/etc/udev/rules.d`" directory and to restart your system.

```
# udev rules to allow write access to all users
# for Yoctopuce USB devices
SUBSYSTEM=="usb", ATTR{idVendor}=="24e0", MODE="0666"
```

Example 2: 51-yoctopuce_group.rules

This rule authorizes the "yoctogroup" group to have read and write access to Yoctopuce USB peripherals. Access rights for all other peripherals are not modified. If this scenario suits you, you only need to copy the "`51-yoctopuce_group.rules`" file into the "`/etc/udev/rules.d`" directory and restart your system.

```
# udev rules to allow write access to all users of "yoctogroup"
# for Yoctopuce USB devices
SUBSYSTEM=="usb", ATTR{idVendor}=="24e0", MODE="0664", GROUP="yoctogroup"
```

25.4. ARM Platforms: HF and EL

There are two main flavors of executable on ARM: HF (Hard Float) binaries, and EL (EABI Little Endian) binaries. These two families are not compatible at all. The compatibility of a given ARM platform with of one of these two families depends on the hardware and on the OS build. ArmHL and ArmEL compatibility problems are quite difficult to detect. Most of the time, the OS itself is unable to make a difference between an HF and an EL executable and will return meaningless messages when you try to use the wrong type of binary.

All pre-compiled Yoctopuce binaries are provided in both formats, as two separate ArmHF et ArmEL executables. If you do not know what family your ARM platform belongs to, just try one executable from each family.

25.5. Powered module but invisible for the OS

If your Yocto-RS485 is connected by USB, if its blue led is on, but if the operating system cannot see the module, check that you are using a true USB cable with data wires, and not a charging cable. Charging cables have only power wires.

25.6. Another process named xxx is already using yAPI

If when initializing the Yoctopuce API, you obtain the "*Another process named xxx is already using yAPI*" error message, it means that another application is already using Yoctopuce USB modules. On a single machine only one process can access Yoctopuce modules by USB at a time. You can easily work around this limitation by using a VirtualHub and the network mode [4].

25.7. Disconnections, erratic behavior

If you Yocto-RS485 behaves erratically and/or disconnects itself from the USB bus without apparent reason, check that it is correctly powered. Avoid cables with a length above 2 meters. If needed, insert a powered USB hub [5] [6].

[4] see: http://www.yoctopuce.com/EN/article/error-message-another-process-is-already-using-yapi
[5] see: http://www.yoctopuce.com/EN/article/usb-cables-size-matters
[6] see: http://www.yoctopuce.com/EN/article/how-many-usb-devices-can-you-connect

25.8. Registering a VirtualHub disconnect an other one

If, when performing a call to RegisterHub() with an VirtualHub address, an other previously registered VirtualHub disconnects, make sure the machine running theses VirtualHubs don't have the same *Hostname*. Same *Hostname* can happen very easily when the operating system is installed from a monolithic image, Raspberry-PI are the best example. The Yoctopuce API uses serial numbers to communicate with devices and VirtualHub serial number are created on the fly based the *hostname* of the machine running the VirtualHub.

25.9. Dropped commands

If, after sending a bunch of commands to a Yoctopuce device, you are under the impression that the last ones have been ignored, typical example is a quick and dirty program meant to configure a device, make sure you used a YAPI.FreeAPI() at the end of the program. Commands are sent to Yoctopuce modules asynchronously thanks to a background thread. When the main program terminates, that thread is killed no matter if some command are left to be sent. However API.FreeAPI () will wait until there is no more commands to send before freeing the API resources and returning.

25.10. Damaged device

Yoctopuce strives to reduce the production of electronic waste. If you believe that your Yocto-RS485 is not working anymore, start by contacting Yoctopuce support by e-mail to diagnose the failure. Even if you know that the device was damaged by mistake, Yoctopuce engineers might be able to repair it, and thus avoid creating electronic waste.

Waste Electrical and Electronic Equipment (WEEE) If you really want to get rid of your Yocto-RS485, do not throw it away in a trash bin but bring it to your local WEEE recycling point. In this way, it will be disposed properly by a specialized WEEE recycling center.

26. Characteristics

You can find below a summary of the main technical characteristics of your Yocto-RS485 module.

Product ID	RS485MK1
Hardware release†	
USB connector	micro-B
Width	20 mm
Length	56 mm
Weight	8 g
Chipset	ADM2587E
Max frequency	115200 bps
Buffer	16 KB
Protection class, according to IEC 61140	class III
USB insulation, dielectric strength (1 min.)	2.5 kV
Normal operating temperature	5...40 °C
Extended operating temperature‡	-30...85 °C
RoHS compliance	RoHS III (2011/65/UE+2015/863)
USB Vendor ID	0x24E0
USB Device ID	0x0048
Suggested enclosure	YoctoBox-Long-Thick-Black
Harmonized tariff code	8542.3190
Made in	Switzerland

† These specifications are for the current hardware revision. Specifications for earlier revisions may differ.

‡ The extended temperature range is defined based on components specifications and has been tested during a limited duration (1h). When using the device in harsh environments for a long period of time, we strongly advise to run extensive tests before going to production.

All dimensions are in mm
Toutes les dimensions sont en mm

Yocto-RS485

A4

Scale
2:1
Echelle