



Yocto-RS232-C

User's guide

Table of contents

1. Introduction	1
1.1. <i>Safety Information</i>	2
1.2. <i>Environmental conditions</i>	3
2. Presentation	5
2.1. <i>Common elements</i>	5
2.2. <i>Specific elements</i>	6
2.3. <i>Functional isolation</i>	7
2.4. <i>Optional accessories</i>	8
3. First steps	11
3.1. <i>Prerequisites</i>	11
3.2. <i>Testing USB connectivity</i>	12
3.3. <i>Localization</i>	13
3.4. <i>Test of the module</i>	13
3.5. <i>Configuration</i>	14
4. Assembly and connections	17
4.1. <i>Fixing</i>	17
4.2. <i>USB power distribution</i>	18
4.3. <i>Electromagnetic compatibility (EMI)</i>	18
5. The serial port	21
5.1. <i>Configurable parameters</i>	21
5.2. <i>Text-line based protocol</i>	22
5.3. <i>Text protocol delimited by STX and ETX</i>	22
5.4. <i>Frame-based binary protocol</i>	23
5.5. <i>MODBUS protocol</i>	23
5.6. <i>Wiegand Protocol</i>	24
5.7. <i>ASCII data stream</i>	24
5.8. <i>Binary data stream</i>	24
5.9. <i>Serial communication analyzer</i>	25
6. Automatic measures	27

6.1. Communication jobs	27
6.2. Tasks	28
6.3. Commands	30
6.4. The genericSensor functions	34
6.5. Configuration examples	35
7. Programming, general concepts	41
7.1. Programming paradigm	41
7.2. The Yocto-RS232-C module	43
7.3. Module	44
7.4. SerialPort	45
7.5. GenericSensor	47
7.6. DataLogger	48
7.7. Files	49
7.8. What interface: Native, DLL or Service ?	50
7.9. Accessing modules through a hub	52
7.10. Programming, where to start?	53
8. Using the Yocto-RS232-C in command line	55
8.1. Installing	55
8.2. Use: general description	55
8.3. Control of the SerialPort function	56
8.4. Control of the module part	57
8.5. Limitations	57
9. Using the Yocto-RS232-C with Python	59
9.1. Source files	59
9.2. Dynamic library	59
9.3. Control of the SerialPort function	59
9.4. Control of the module part	61
9.5. Error handling	63
10. Using Yocto-RS232-C with C++	65
10.1. Control of the SerialPort function	65
10.2. Control of the module part	67
10.3. Error handling	70
10.4. Integration variants for the C++ Yoctopuce library	70
11. Using Yocto-RS232-C with C#	73
11.1. Installation	73
11.2. Using the Yoctopuce API in a Visual C# project	73
11.3. Control of the SerialPort function	74
11.4. Control of the module part	76
11.5. Error handling	78
12. Using the Yocto-RS232-C with LabVIEW	81
12.1. Architecture	81
12.2. Compatibility	82
12.3. Installation	82
12.4. Presentation of Yoctopuce VIs	87
12.5. Functioning and use of VIs	90
12.6. Using	92

12.7. Managing the data logger	94
12.8. Function list	95
12.9. A word on performances	96
12.10. A full example of a LabVIEW program	96
12.11. Differences from other Yoctopuce APIs	97
13. Using the Yocto-RS232-C with Java	99
13.1. Getting ready	99
13.2. Control of the SerialPort function	99
13.3. Control of the module part	101
13.4. Error handling	104
14. Using the Yocto-RS232-C with Android	105
14.1. Native access and VirtualHub	105
14.2. Getting ready	105
14.3. Compatibility	105
14.4. Activating the USB port under Android	106
14.5. Control of the SerialPort function	107
14.6. Control of the module part	110
14.7. Error handling	114
15. Using Yocto-RS232-C with TypeScript	117
15.1. Using the Yoctopuce library for TypeScript	118
15.2. Refresher on asynchronous I/O in JavaScript	118
15.3. Control of the SerialPort function	119
15.4. Control of the module part	122
15.5. Error handling	124
16. Using Yocto-RS232-C with JavaScript / EcmaScript	127
16.1. Blocking I/O versus Asynchronous I/O in JavaScript	127
16.2. Using Yoctopuce library for JavaScript / EcmaScript 2017	128
16.3. Control of the SerialPort function	130
16.4. Control of the module part	134
16.5. Error handling	136
17. Using Yocto-RS232-C with PHP	139
17.1. Getting ready	139
17.2. Control of the SerialPort function	140
17.3. Control of the module part	142
17.4. HTTP callback API and NAT filters	144
17.5. Error handling	148
18. Using Yocto-RS232-C with Visual Basic .NET	149
18.1. Installation	149
18.2. Using the Yoctopuce API in a Visual Basic project	149
18.3. Control of the SerialPort function	150
18.4. Control of the module part	152
18.5. Error handling	154
19. Using Yocto-RS232-C with Delphi or Lazarus	155
19.1. Preparation	155
19.2. About examples	156

19.3. Control of the SerialPort function	156
19.4. Control of the module part	158
19.5. Error handling	160
20. Using the Yocto-RS232-C with Universal Windows Platform	163
20.1. Blocking and asynchronous functions	163
20.2. Installation	164
20.3. Using the Yoctopuce API in a Visual Studio project	164
20.4. Control of the SerialPort function	165
20.5. A real example	166
20.6. Control of the module part	167
20.7. Error handling	169
21. Using Yocto-RS232-C with Objective-C	171
21.1. Control of the SerialPort function	171
21.2. Control of the module part	173
21.3. Error handling	175
22. Using with unsupported languages	177
22.1. Command line	177
22.2. .NET Assembly	177
22.3. VirtualHub and HTTP GET	179
22.4. Using dynamic libraries	181
22.5. Porting the high level library	184
23. Advanced programming	185
23.1. Event programming	185
23.2. The data logger	188
23.3. Sensor calibration	190
24. Firmware Update	195
24.1. VirtualHub or the YoctoHub	195
24.2. The command line library	195
24.3. The Android application Yocto-Firmware	195
24.4. Updating the firmware with the programming library	196
24.5. The "update" mode	198
25. High-level API Reference	199
25.1. Class YAPI	200
25.2. Class YModule	204
25.3. Class YSerialPort	211
25.4. Class YFiles	220
25.5. Class YGenericSensor	225
25.6. Class YDataLogger	233
25.7. Class YDataSet	238
25.8. Class YMeasure	241
26. Troubleshooting	243
26.1. Where to start?	243
26.2. Programming examples don't seem to work	243
26.3. Linux and USB	243
26.4. ARM Platforms: HF and EL	244

26.5. Powered module but invisible for the OS	244
26.6. Another process named xxx is already using yAPI	244
26.7. Disconnections, erratic behavior	244
26.8. After a failed firmware update, the device stopped working	245
26.9. Registering VirtualHub disconnects another instance	245
26.10. Dropped commands	245
26.11. Damaged device	245
27. Characteristics	247
<i>Blueprint</i>	249

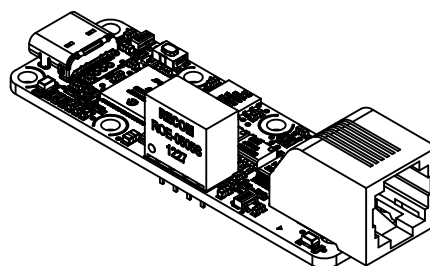
1. Introduction

The Yocto-RS232-C is a 60x20mm USB module with an RS232 serial port. It natively supports several serial protocols, including MODBUS. Its buffer memory enables it to communicate asynchronously, if need be. The Yocto-RS232-C can also work as a serial communication analyzer. In the opposite to most common USB/serial adaptors, it does not need drivers and, most importantly, its use does not require any virtual COM port.

On top of offering low level serial communications, the Yocto-RS232-C can autonomously question and analyze the serial output of any appliance to then present the results in the manner of a Yoctopuce sensor. In other words, the Yocto-RS232-C can transform any sensor equipped with a serial output into the software equivalent of a Yoctopuce sensor, data logger included.

An important characteristic of the Yocto-RS232-C is to be an isolated module: the communication part is electrically isolated from the USB part. This enables the module, for example, to be connected to devices powered by the mains without risking to destroy your computer.

Beware, the Yocto-RS232-C is not a classic serial to USB adaptor: it does not create a virtual COM port and you cannot therefore use it with an application designed to use a COM port.



The Yocto-RS232-C module

The Yocto-RS232-C 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-RS232-C 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-RS232-C 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:	Route de Cartigny 33
ZIP code, city:	1236 Cartigny
Country:	Switzerland

1.1. Safety Information

The Yocto-RS232-C 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

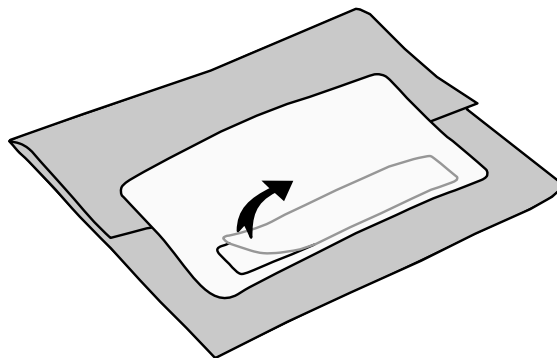
The Yocto-RS232-C should not be used without a protective enclosure, because of the accessible bare electronic components. For optimal safety, it should be put into a non-metallic, non-inflammable enclosure, resistant to a mechanical stress level of 5 J. For instance, use a polycarbonate (e.g. LEXAN) 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 stick the water-resistant identification label provided together with the electronic board as close as possible to the device. If the device is put in a dedicated enclosure, the identification label should be affixed on the outside of the enclosure. This label is resistant to humidity and to the usual rubbing that can occur during normal maintenance.



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-RS232-C for another kind of application, you should check the safety regulations according to the standard applicable to your application.

In particular, the Yocto-RS232-C is *not* certified for use in medical environments or for life-support applications.

Environment

The Yocto-RS232-C 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-RS232-C 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-RS232-C terminal blocks any wire that could be connected to the mains.

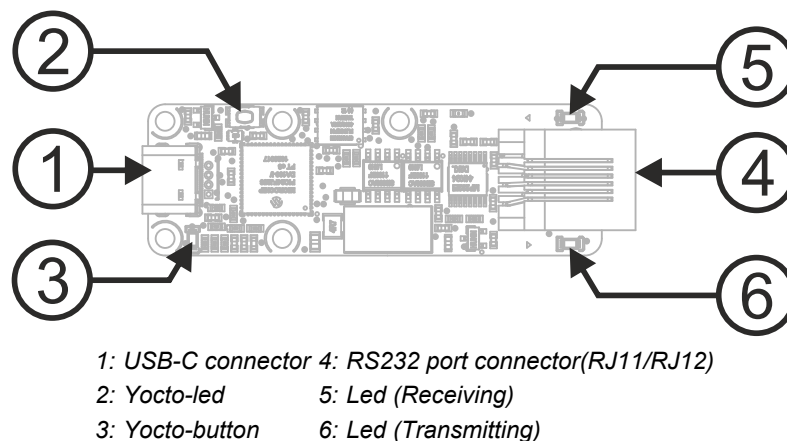
1.2. Environmental conditions

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. Yoctopuce's 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-RS232-C is -25...85°C. This temperature range has been determined based on components manufacturer recommendations, and on controlled environment tests performed during a limited duration (1h). If you plan to use the Yocto-RS232-C in harsh environments for a long period of time, we strongly advise you to run extensive tests before going to production.

2. Presentation



2.1. Common elements

All Yocto-modules share a number of common functionalities.

USB connector

The Yocto-RS232-C features an USB Type-C socket and uses USB 1.1 protocol which makes it compatible with any USB host port. Alternatively USB wires or a 1.27mm connector can be soldered on the footprint right behind the USB socket.

If, when you plug your Yocto-RS232-C into a USB-C cable, your module lights up but is not detected by your computer, check that the USB-C plug is fully inserted and that you're using a regular USB cable, not a charging cable with no data lines.

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

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

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 support².

Current sensor

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

Serial number

Each Yocto-module has a unique serial number assigned to it at the factory. For Yocto-RS232-C modules, this number starts with RS232M1C. 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

RS232 port connector

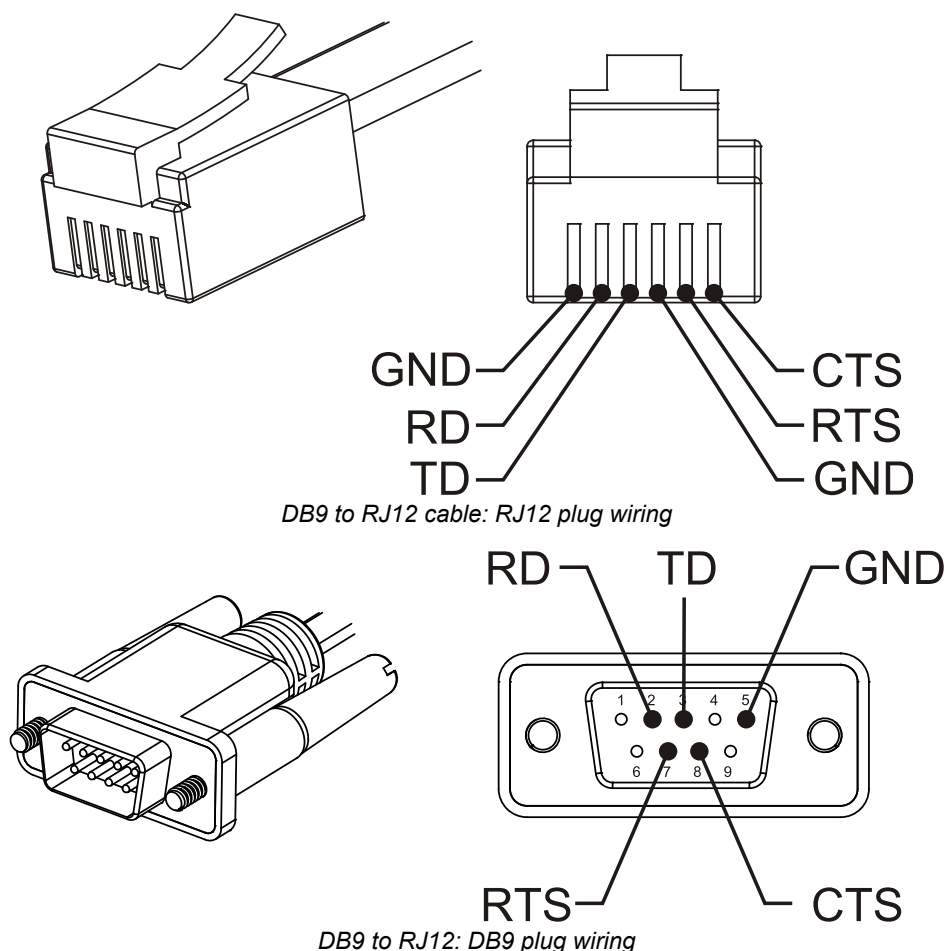
The Yocto-RS232-C modules contains a serial port to the RS232 standard³. In the opposite to classic RS232 adaptors, the Yocto-RS232-C is not a simple gateway to a virtual COM port with an identity varying over time. Instead, the Yocto-RS232-C autonomously manages the serial port: it contains 16KB input and output buffers, it automatically manages flow control, and it can recognize some usual message formats (including MODBUS formats) to ease reading by software. You can find more details in the chapter of this documentation devoted to the serial port. For reasons of space, the serial port connector follows the RJ11 / RJ12 standard. To use the Yocto-RS232-C with a standard

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

² support@yoctopuce.com

³ The precise name of the standard is TIA/EIA-232-F.

RS232 connector, you need a "DB-9"⁴ to RJ11 or RJ12 cable. If you need flow control, you must use a male DB9 to RJ12 (6P6C) cable. If you do not require hardware flow control (CTS/RTS), you can content yourself with a RJ11 connector (6P4C, that is without the two outside contacts). As there is no standard for this kind of cables, pin assignment is specific to this Yoctopuce module.



The communication circuit is a safety extra low voltage (SELV) circuit. It should not be presented with voltages exceeding 60 VDC, nor connected to mains circuits.

Activity leds

The Yocto-RS232-C contains two green leds reflecting the RS232 port activity. The receiving led lights up feebly when a CTS signal is received (RTS line activated by the external device). It lights up strongly when it receives a character. The transmitting led lights up feebly when the module sends the RTS signal and strongly when it sends a character.

2.3. Functional isolation

The Yocto-RS232-C is designed as two distinct electrical circuits, separated by a functional isolation. This isolation plays no role for the operator safety, since both circuits of the Yocto-RS232-C work with safety extra low voltages (SELV) and are accessible without risk at any time. The isolation has been added in excess of safety requirements, to improve the reliability and the ease of use of the Yocto-RS232-C, allowing both circuits to work with different reference grounds.

⁴ The plug usually called "DB-9" is a D-Sub 9 male pin plug. Its correct denomination is DE-9.

Although the isolation plays no role for security, it has been designed according to the rules that would apply for a supplementary isolation on a secondary circuit. Its specifications of the functional isolation are as follows⁵:

- **Isolation voltage**⁶: 1kV
- **Clearance distance**: 1mm
- **Creepage distance**: 1mm
- **Material group**: Cat IIIa (FR4)

2.4. Optional accessories

The accessories below are not necessary to use the Yocto-RS232-C module but might be useful depending on your project. These are mostly common products that you can buy from your favorite DIY 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-RS232-C 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 multi-TT 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-RS232-C, thanks to the YoctoHub-Ethernet, the YoctoHub-Wireless and the YoctoHub-GSM which provides respectively Ethernet, WiFi and GSM connectivity. All of them can drive up to three devices and behave exactly like a regular computer running the VirtualHub application⁷.

1.27mm (or 1.25mm) connectors

In case you wish to connect your Yocto-RS232-C 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-RS232-C 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-RS232-C.

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

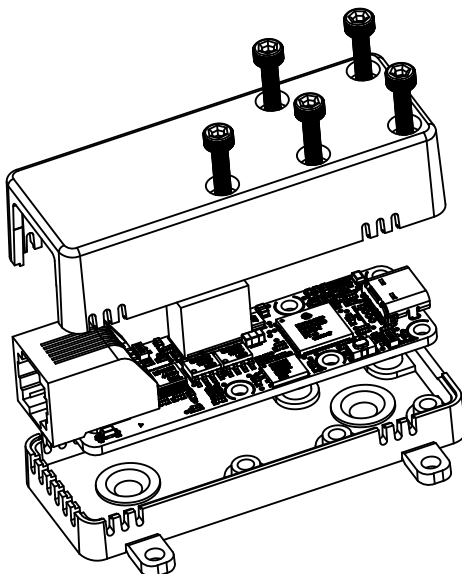
Your Yocto-RS232-C has been designed to be installed as is in your project. Nevertheless, Yoctopuce sells enclosures specifically designed for Yoctopuce devices. These enclosures have removable mounting brackets and magnets allowing them to stick on ferromagnetic surfaces. More

⁵ This description of the isolation 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.

⁶ **Nominal value, not tested**

⁷ <http://www.yoctopuce.com/EN/virtualhub.php>

details are available on the Yoctopuce web site⁸. The suggested enclosure model for your Yocto-RS232-C is the YoctoBox-Long-Thick-Black-RS232.



You can install your Yocto-RS232-C in an optional enclosure

Analyseur de communication sÃ©rie

The Yocto-RS232-C can be used as a serial protocol analyzer by connecting it on the cable that directly connects two devices communicating via a serial protocol. The wiring required to do so is described in the manual, but if you prefer 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-Adapter⁹.

⁸ <http://www.yoctopuce.com/EN/products/category/enclosures>

⁹ <http://www.yoctopuce.com/EN/products/accessories-and-connectors/rs232-snooping-adapter>

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-RS232-C 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, Linux, macOS, and Android**. Yoctopuce modules do not require the installation of specific drivers, as they use the HID driver¹ standardly supplied in all operating systems.

The general rule regarding supported operating system versions is as follows: Yoctopuce development tools are supported for all versions covered by the operating system vendor's support, including the duration of extended support (*long term support* or LTS). Yoctopuce pays particular attention to long-term support, and whenever possible with reasonable effort, our tools are designed so that they can be used on older systems even several years after the end of the manufacturer's extended support.

Moreover, the programming libraries used to drive our modules being available in source code, you can generally recompile them to run on even older operating systems. To date, our programming library can still be compiled to run on operating systems released in 2008, such as Windows XP SP3 or Linux Debian Squeeze.

The architectures supported by Yoctopuce software libraries are as follows:

- Windows: Intel 64 bits and 32 bits
- Linux: Intel 64 bits and 32 bits, ARM 64 bits and 32 bits, including Raspberry Pi OS.
- macOS: Intel 64 bits and Apple Silicon (ARM)

Under Linux, communication with our USB modules requires the libusb library, version 1.0 or higher, which is available on all common distributions. Libraries and command-line tools should be easy to

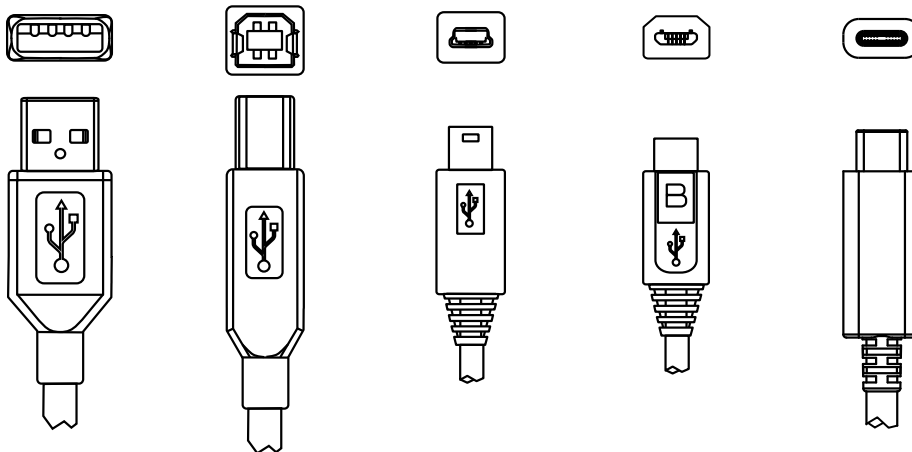
¹ The HID driver is manages peripheral devices such as mouse, keyboard, and so on.

recompile on any UNIX variant (Linux, FreeBSD, ...) from the last fifteen years for which libusb-1.0 is available and functional.

Under Android, the ability to connect a USB module depends on whether the tablet or phone supports the *USB Host mode*.

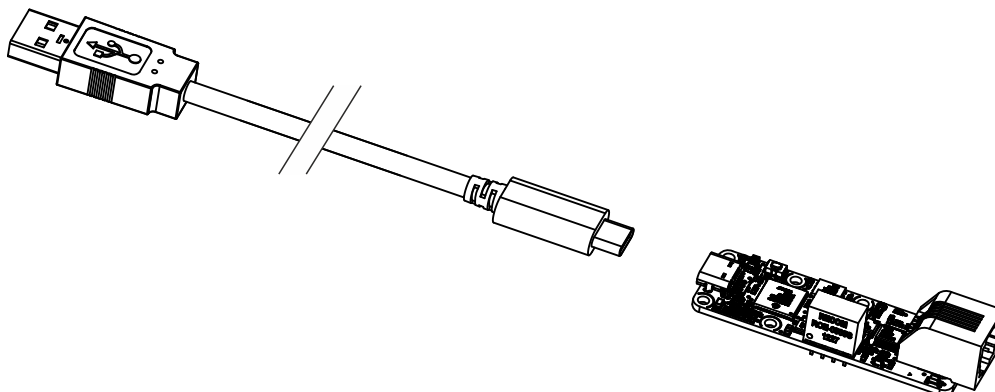
An USB cable, type A - USB-C

USB connectors come in several shapes. The "standard" size is the one you probably use to connect your printer. The "mini" size has more or less disappeared. The "micro" size was the smallest when the first Yoctopuce modules were designed. Over the last few years, USB-C connectors have appeared and are about to replace all other types. That is why, since 2024 Yoctopuce, is progressively migrating its product line to USB-C².



The most common USB 2.0 connectors: A, B, Mini B, Micro B and USB-C.

To connect your Yocto-RS232-C module to a computer, you need an USB cable of type A-USB-C or type C-USB-C. The price of this cable may vary a lot depending on the source, look for it under the name *USB A to USB-C 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-RS232-C module with a USB 2.0 cable of type A-USB-C

If you insert a USB hub between the computer and the Yocto-RS232-C 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-RS232-C should be connected to your computer, which should have recognized it. It is time to make it work.

² www.yoctopuce.com/EN/article/would-you-like-usb-c-devices

Go to the Yoctopuce web site and download the *Virtual Hub* software³. It is available for Windows, Linux, and macOS. Normally, VirtualHub serves as an abstraction layer for languages which cannot access the hardware layers of your computer. However, it also offers a succinct interface to configure your modules and to test their basic functions. You access this interface with a simple web browser⁴. Start VirtualHub in a command line, open your preferred web browser and enter the URL `http://127.0.0.1:4444`. The list of the Yoctopuce modules connected to your computer is displayed.

Serial	Logical Name	Description	Action
VIRTHUB2-3880db7f12		VirtualHub-V2	configure view log file
↳ RS232M1C-2C1D38		Yocto-RS232-C	configure view log file beacon

[Show debug information](#)
[Show device functions](#)

Module list as displayed in your web browser

3.3. Localization

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

3.4. Test of the module

The first item to check is that your module is working well: click on the serial number corresponding to your module. This displays a window summarizing the properties of your Yocto-RS232-C.

RS232M1C-2C1D38



RS232M1C-2C1D38 is a 60x20mm board featuring an isolated RS232 interface with 16KB buffer, message analyzer and a built-in datalogger.

Module

Serial #

Product name

Logical name

Firmware

Consumption

Beacon

Luminosity

RS232M1C-2C1D38

Yocto-RS232-C

65297

58 mA

Inactive

50%

turn on

Serial interface

Protocol

Current job

Send

Line (9600,8N1)

no job active

change

send

Time

Message

Job

export

clear

pause

view log

Misc

Open API browser

Get user manual from yoctopuce.com

Close

Properties of the Yocto-RS232-C 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

³ www.yoctopuce.com/EN/virtualhub.php

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

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.

RS232M1C-2C1D38

Edit parameters for device RS232M1C-2C1D38, and click on the **Save** button.

Serial #: RS232M1C-2C1D38
 Product name: Yocto-RS232-C
 Firmware: 65297 Upgrade

Logical name:
 Luminosity: (signal leds only)

Device functions

Each function of the device has a physical name and a logical name. You can change the logical name using the **rename** button. You can also configure the serial transmission parameters, as well as the way serial data is analyzed to be notified to the host.

Physical Name	Logical Name	Action
RS232M1C-2C1D38.files /		manage files
User files: 0 file, 472 KB available		
RS232M1C-2C1D38.genericSensor1 /		configure
No mapping configured		
RS232M1C-2C1D38.genericSensor2 /		configure
No mapping configured		
RS232M1C-2C1D38.genericSensor3 /		configure
No mapping configured		
RS232M1C-2C1D38.genericSensor4 /		configure
No mapping configured		
RS232M1C-2C1D38.genericSensor5 /		configure
No mapping configured		
RS232M1C-2C1D38.genericSensor6 /		configure
No mapping configured		
RS232M1C-2C1D38.genericSensor7 /		configure
No mapping configured		
RS232M1C-2C1D38.genericSensor8 /		configure
No mapping configured		
RS232M1C-2C1D38.genericSensor9 /		configure
No mapping configured		
RS232M1C-2C1D38.genericSensor10 /		configure
No mapping configured		
RS232M1C-2C1D38.serialPort /		configure
Type of protocol:	Line-based ASCII protocol	
Baud rate and encoding:	9600 8N1	
Flow control:	none	
Min. interval between bytes sent (ms):	0	
No job file defined		manage job files
Startup job:	no startup job	
RS232M1C-2C1D38.dataLogger /		configure

Save **Cancel**

Yocto-RS232-C 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 VirtualHub ⁵ in command line ⁶.

Logical name of the module

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

Luminosity

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

Logical names of functions

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

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 the flow control.

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

⁵ www.yoctopuce.com/EN/virtualhub.php

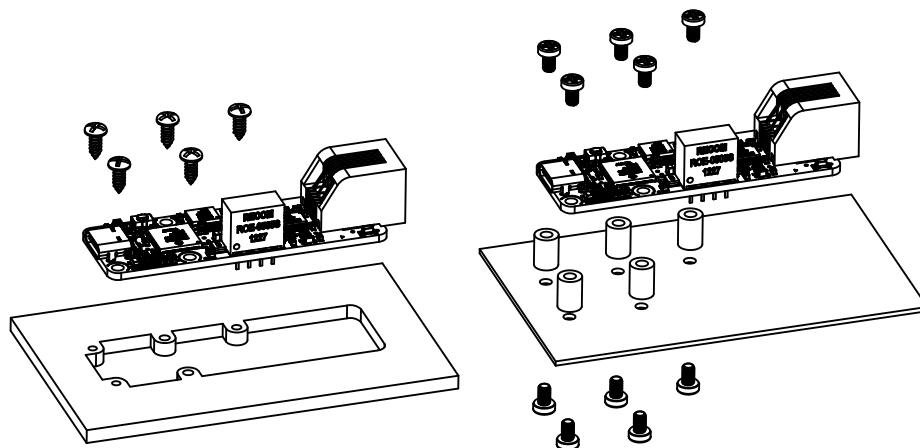
⁶ More information available in the VirtualHub documentation

4. Assembly and connections

This chapter provides important information regarding the use of the Yocto-RS232-C 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-RS232-C module contains 2.5mm assembly holes. You can use these holes for screws. The screw head diameter must not be larger than 4.5mm or they will damage the module circuits. Make sure that the lower surface of the module is not in contact with the support. We recommend using spacers, but other methods are possible. Nothing prevents you from fixing the module with a glue gun; it will not be good-looking, but it will hold.

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

4.2. USB power distribution

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

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

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

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

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

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

4.3. Electromagnetic compatibility (EMI)

Connection methods to integrate the Yocto-RS232-C obviously have an impact on the system overall electromagnetic emissions, and therefore also impact the conformity with international standards.

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

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

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

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

5. The serial port

In the opposite to classic RS232 adapters, the Yocto-RS232-C 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 autonomously manages flow control, for hardware (CTS/RTS) as well as for software (XON/XOFF)
- it supports driving CTS/RTS lines in half-duplex mode for the management of a shared serial bus
- 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-RS232-C 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-RS232-C RS232 port follows the TIA/EIA-232-F standard. Generated signals have a +/- 5V level, which are correctly decoded by all RS232 interfaces conforming to the standard. Moreover, the module accepts input levels up to +/- 30V.

5.1. Configurable parameters

The Yocto-RS232-C 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

Depending on the configuration of the device connected to the serial port, you can enable a flow control function in the Yocto-RS232-C. Hardware flow control (based on CTS/RTS lines) is the most

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.

efficient, but it is not available on all devices.² Software flow control (based on sending XON/XOFF codes) is also supported.

There is a third variant called "Simplex", to support transmissions in *half-duplex* where only one interface can speak at a time: The RTS line is signaled when the Yocto-RS232-C must transmit information on the serial bus, and the transmission happens only when the CTS line confirms that the bus is available. The CTS line is released at the end of the transmission. This mode is used by some RS232 to RS485 adapters, among others.

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.

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.

² Beware, if hardware flow control is enabled in the Yocto-RS232-C but that the CTS line is either not wired on or not managed by the connected device, transmission is blocked. Therefore, flow control is not enabled in the default configuration.

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-RS232-C 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-RS232-C 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-RS232-C 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-RS232-C 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:

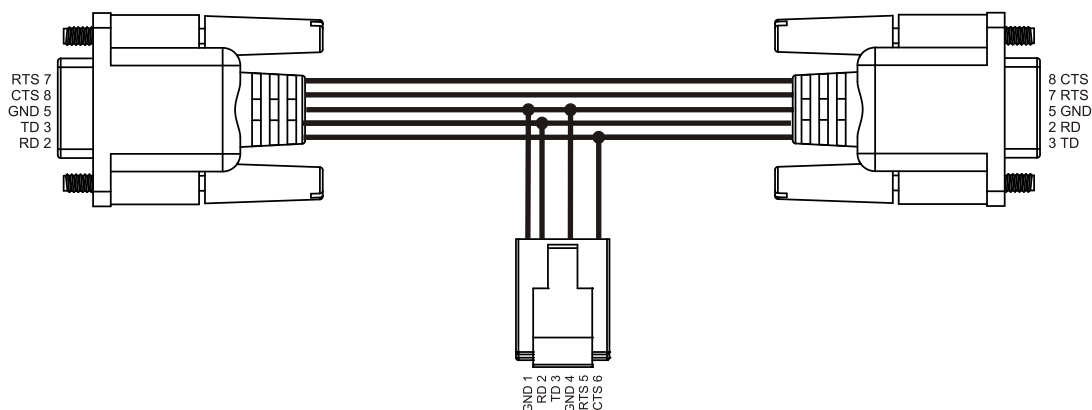
- **reset()**, to empty the incoming buffer
- **writeHex()** to send a byte string (given a hexadecimal string)
- **readHex()** to read a byte string (given as a hexadecimal string)
- **read_seek()** to move the read position pointer within the buffer

- `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-RS232-C 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-RS232-C 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-RS232-C. It is then able to read the traffic going through both ways on the serial cable, identifying the direction of the communication.



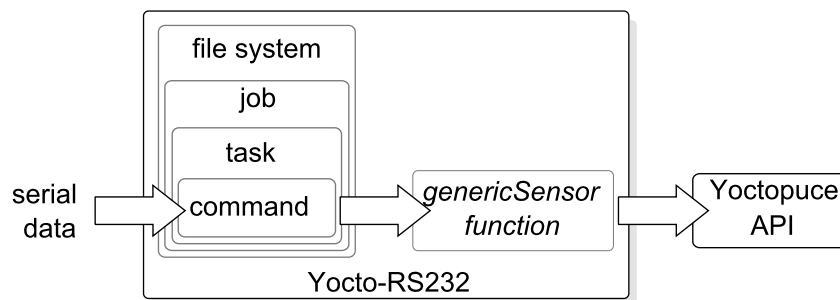
Wiring to use the Yocto-RS232-C in analyzer mode

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-Adapter³.

³ <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-RS232-C 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-RS232-C can automatically send and receive data on the serial port.

6.1. Communication jobs

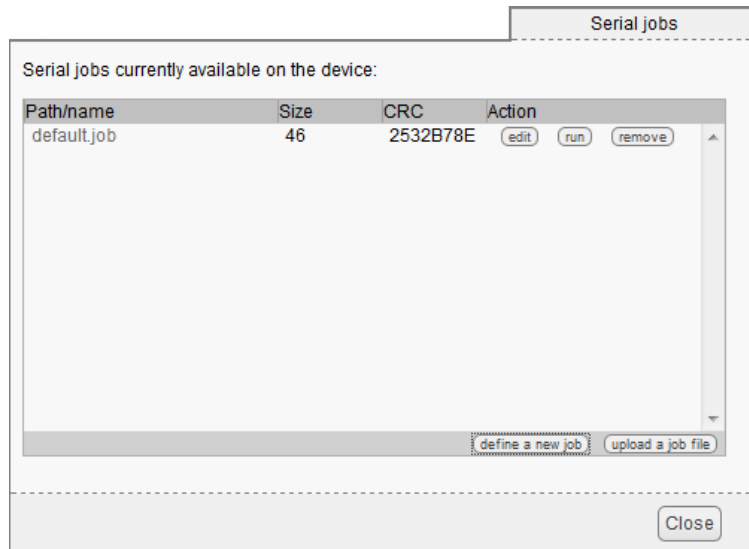
The Yocto-RS232-C contains a file system in which you can store jobs, which are in fact simple text files in the JSON format. A job describes write and read actions to be performed on the serial port. In the VirtualHub interface, the window describing the Yocto-RS232-C properties allows you to select which job must be run, while the configuration window enables you to select which job must be run when the module starts. The Yocto-RS232-C 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.

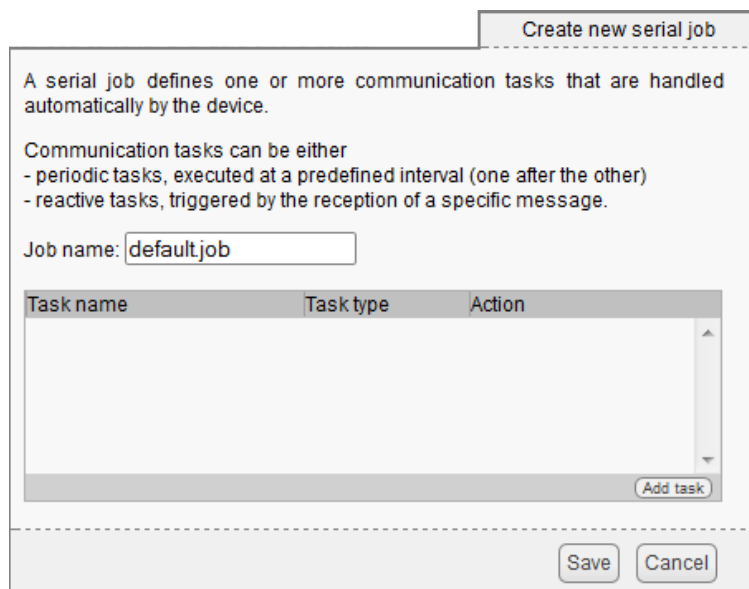
Job definition and management

You can define jobs with the VirtualHub, in the Yocto-RS232-C 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-RS232-C. To configure the Yocto-RS232-C by software, you simply need to upload the correct file on the Yocto-RS232-C 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-RS232-C 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-RS232-C: 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-RS232-C. 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-RS232-C *genericSensor* functions. From the developer stand point, the device that is connected to the Yocto-RS232-C 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-RS232-C configuration must correspond to the needs of the job. For instant, you cannot detect a MODBUS transaction if the Yocto-RS232-C 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-RS232-C initiative. They are generally used to send commands to the device connected to the Yocto-RS232-C. 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-RS232-C *genericSensor* functions. Beware, the serial protocol defined in the Yocto-RS232-C configuration must correspond to the needs of the job: for example, you cannot detect a MODBUS transaction if the Yocto-RS232-C is configured in *line-based ASCII* mode.

The screenshot shows a dialog box titled "Edit task" with a dashed border. It contains four numbered sections for configuring a task:

- 1. Select a name and a type for your task:**
 - Name:
 - Type:
 - ☐ Reactive task (triggered by data received on the serial port)
 - ☒ Periodic task (sending data at predefined interval)
 - [Advanced properties](#)
- 2. Choose a standard periodic task, or define your own protocol**
 - ☒ Send a simple text
 - ☐ Send binary data
 - ☐ Write to MODBUS device
 - ☐ Read from MODBUS device
 - ☐ Read from SCPI instrument
 - ☐ Use a custom protocol
- 3. Enter the text that you want to send:**
 -
- 4. Enter the time interval between repeated executions of this task:**
 - ☒ Repeat interval: ms
 - ☐ single run

At the bottom right, there are "Ok" and "Cancel" buttons.

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-RS232-C 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 bytes 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 "($temp > 0 ? log($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.

Two time functions are available.

utctime()	returns the number of seconds elapsed since midnight UTC.
utctime()	returns the number of days elapsed since January 1, 1970.

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
```

You can use the `utctime()` and `utcdate()` time functions in an `assert` command to ensure that a task only runs when a specific time condition is met, such as on the full hour.

```
assert utctime() mod 3600 == 0
```

WAIT

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

SYNC

The *sync* command waits until the next multiple of the specified period passes in milliseconds. The reference time is UTC.

LOG

The *log* command displays a character string in the logs of the Yocto-RS232-C. You can use the same substitutions as described for the *writeVar* command.

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)

WRITEVAR

The *writeVar* command behaves like the *write* command (or *writeLine* on the Yocto-I2C), but first makes substitutions in the string sent, replacing the following expressions:

- **(\$x:INT)** sends the value of variable \$x as an integer (decimal).
- **(\$x:FLOAT)** sends the value of variable \$x as a real number.
- **(\$x:BYTE)** sends the value of variable \$x as a hexadecimal byte (two hexadecimal digits).
- **(\$x:WORD)** and **(\$x:WORDL)** send the value of variable \$x as a hexadecimal word (four hexadecimal digits). The L suffix is used to send the value in little-endian format, i.e. with the most significant byte at the end.

- **(\$x:DWORD)** and **(\$x:DWORDL)** are used to send the value of variable \$x as a hexadecimal dword (eight hexadecimal digits). The L suffix is used to send the value in little-endian format, i.e. with the most significant byte at the end.
- **(\$x:FLOAT32B)** and **(\$x:FLOAT32L)** send the value of the variable \$x in hexadecimal as a 32-bit IEEE 754 floating-point number, respectively with the bytes ordered as big-endian or little-endian.

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-RS232-C.

6.4. The genericSensor functions

9 *genericSensor* functions are available in the Yocto-RS232-C. 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 examples

Here are a few job examples to interface commercial devices.

DYLOS, Air quality monitor, DC1100 PRO

The DC1100 PRO sensor is a dust counter that is used to estimate air quality. It has a serial port on which it sends once per minute two numbers separated by commas:

- The number of small particles per cubic foot
- The number of large particles per cubic foot in the last 10 seconds

For example:

```
823,57
854,65
850,63
```

Step 1

Configure the Yocto-RS232-C in "Line-based ASCII protocol", 9600, 8N1, without flow control.

Step 2

Create a job containing a single reactive task, waiting for a CVS record, with a comma as character separator, with field 1 and field 2 assigned to the *genericSensor1* et *genericSensor2* functions.

Edit task

1. Select a name and a type for your task:

Name:

Type: ☒ Reactive task (triggered by data received on the serial port)
☐ Periodic task (sending data at predefined interval)

2. Choose a standard reactive task, or define your own protocol

☐ Receive numeric value
☐ Receive GPS NMEA record
☒ Receive CSV record
☐ Detect MODBUS writes
☐ Detect MODBUS reads
☐ Use a custom protocol

3. Specify how to interpret each CSV record:

Field separator:

Add marks in the table below to map sensors to CSV fields:

Sensor \ field index:	1	2	3	4	5	6	7	8	9	10	11	12
genericSensor1	☒	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
genericSensor2	☐	☒	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
genericSensor3	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
genericSensor4	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
genericSensor5	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
genericSensor6	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
genericSensor7	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
genericSensor8	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
genericSensor9	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐

Task example to read a DYLOS DC1100 PRO sensor

Step 3

The measures returned by the DC1100 PRO are expressed in particles per cubic foot. To convert them into particles per cubic meter, you can configure the *genericSensor1* and *genericSensor2* functions to make 0 correspond to 0 and 10000 to 353147 (one m³ = 35.3147 cubic foot). Finally, you can define the unit as P/m³ and the accuracy to 1.

Generic Sensor Configuration

You can map the value fetched from the serial port to a physical measure, proportional to the value read:

Lowest value expected: Map to:

Highest value expected: Map to:

Display unit:

Display resolution:

Automatic conversion of the number of particles per cubic foot into particles per cubic meter

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.

HAMEG HMP2020

The HAMEG HMP2020 is a laboratory power source that you can entirely control through the serial port (SCPI type protocol). You can for example know the delivered current on each channel thanks to three commands :

```
INST OUT1
MEAS:CURR?
SYST:LOCAL
```

Step 1

Configure the Yocto-RS232-C in "Line-based ASCII protocol", 9600, 8N1, without flow control.

Step 2

Create a job containing a single periodic task, sending "*INST OUT1*" to select channel 1, then "*MEAS:CURR?*" to ask for a current measure, and then waits for a decimal number to assign to the `genericSensor1` function, and finally sends "*SYST:LOCAL*" to free the interface.

Edit task

1. Select a name and a type for your task:

Name:

Type: ☐ Reactive task (triggered by data received on the serial port)
☒ Periodic task (sending data at predefined interval)

2. Choose a standard periodic task, or define your own protocol

☐ Send a simple text
☐ Send binary data
☐ Write to MODBUS device
☐ Read from MODBUS device
☐ Read from SCPI instrument
☒ Use a custom protocol

3. Enter the steps that define your custom protocol

Command	Argument	Action
writeLine	INST OUT1	1 I edit delete
writeLine	CURR?	1 I edit delete
expect	(\$1:FLOAT)	1 I edit delete
writeLine	SYST:LOCAL	1 I edit delete

4. Enter the time interval between repeated executions of this task:

Repeat interval: seconds

Task example to read the current on channel 1 of a HAMEG HMP2020 power supply

Step 3

Configure the `genericSensor1` function with `unit=A` and `resolution=0.001`. No need to change the mapping, the values are already returned in the correct unit by the device.

Configuration of the genericSensor function to display the current delivered by the power supply.

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.

Schneider Electric Zelio REG48 Temperature Controller

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

Step 1

Configure the Yocto-RS232-C in "MODBUS-RTU", with speed and parity corresponding to the configuration of your REG48 controller. Note that the REG48 controller works in RS485 and not in RS232: you must therefore use either a Yocto-RS485, or a Yocto-RS232 combined with a RS232/RS485 converter. You cannot connect a Yocto-RS232 directly on a REG48 controller.

Step 2

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

Edit task

1. Select a name and a type for your task:

Name:

Type: ☐ Reactive task (triggered by data received on the serial port)
☒ Periodic task (sending data at predefined interval)

2. Choose a standard periodic task, or define your own protocol

☐ Send a simple text
☐ Send binary data
☐ Write to MODBUS device
☒ Read from MODBUS device
☐ Read from SCPI instrument
☐ Use a custom protocol

3. Enter the MODBUS message details:

MODBUS command: ☐ Read coil
☐ Read input bit
☐ Read holding register
☒ Read input register

MODBUS slave address: (1...255)
 Input register No: (40001...49999)
 Map value to sensor:

4. Enter the time interval between repeated executions of this task:

☒ Repeat interval: ms ☐ single run

Task example to read the temperature on a REG48 MODBUS controller

Step 3

Configure the *genericSensor1* function with unit='C and resolution =0.1. The value obtained from the register is the temperature in 1/10 of degree, so we configure the mapping to divide the value by 10.

Generic Sensor Configuration

You can map the value fetched from the serial port to a physical measure in genericSensor1, proportional to the value read:

Lowest value expected: Map to:
 Highest value expected: Map to:
 Display unit:
 Display resolution:

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

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.

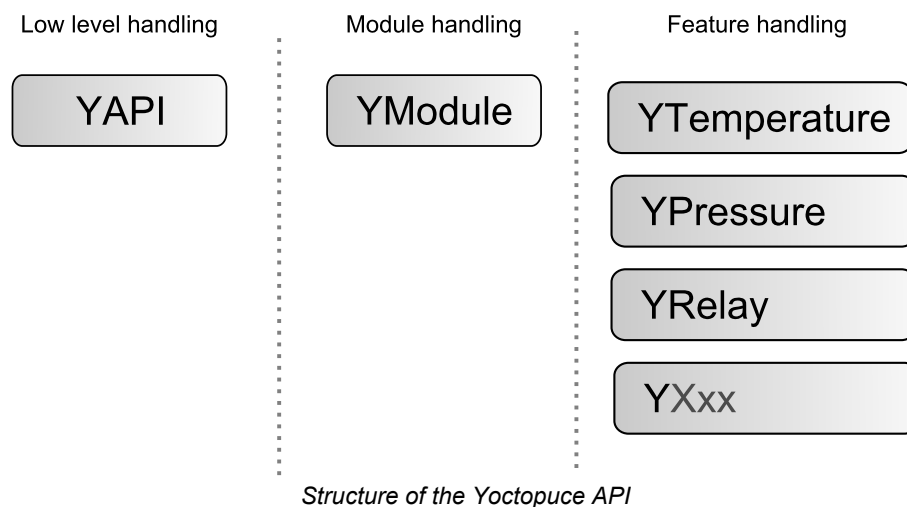
7. Programming, general concepts

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-RS232-C 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*, and so on. 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.



The YSensor class

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

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

Programmation

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

From the programming stand point, your Yocto-RS232-C 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-RS232-C module

The Yocto-RS232-C is an isolated RS232 interface with built-in datalogger.

module : Module

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

serialPort : SerialPort

attribute	type	modifiable ?
logicalName	String	modifiable
advertisedValue	String	modifiable
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	modifiable
startupJob	String	modifiable
jobMaxTask	Integer	read-only
jobMaxSize	Integer	read-only
command	String	modifiable
protocol	Type of messaging protocol	modifiable
voltageLevel	Enumerated	modifiable
serialMode	Serial parameters	modifiable

genericSensor1 : GenericSensor
genericSensor2 : GenericSensor
genericSensor3 : GenericSensor
genericSensor4 : GenericSensor
genericSensor5 : GenericSensor
genericSensor6 : GenericSensor
genericSensor7 : GenericSensor
genericSensor8 : GenericSensor
genericSensor9 : GenericSensor
genericSensor10 : GenericSensor

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

dataLogger : DataLogger

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

files : Files

attribute	type	modifiable ?
logicalName	String	modifiable
advertisedValue	String	modifiable
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-RS232-C module, this serial number always starts with RS232M1C. You can use the serial number to access a given module by software.

logicalName

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

productId

USB device identifier of the module, preprogrammed to 206 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, and so on.

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 a 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-Bridge 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 a non-empty value, it can be used to access the generic sensor directly. If two generic sensors with the same logical name are used in the same project, there is no way to determine which one answers when one tries accessing by logical name. The logical name is limited to 19 characters among `A..Z,a..z,0..9,_`, and `-`.

advertisedValue

Short character string summarizing the current state of the generic sensor, that is automatically advertised up to the parent hub. For a generic sensor, the advertised value is the 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 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 retrieval 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 a non-empty value, it can be used to access the data logger directly. If two data loggers with the same logical name are used in the same project, there is no way to determine which one answers when one tries accessing by logical name. The logical name is limited to 19 characters among A..Z, a..z, 0..9, _, and -.

advertisedValue

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

currentRunIndex

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

timeUTC

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

recording

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

autoStart

Automatic start of the data logger on power on. Setting this attribute ensures that the data logger is always turned on when the device is powered up, without need for a software command. 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 state of the localization beacon and the state of the data logger. If this attribute is set, it is possible to start the recording with the Yocto-button or the attribute `beacon` of the function YModule. In the same way, if the attribute `recording` is changed, the state of the localization beacon is updated. Note: when this attribute is set the localization beacon pulses at a slower rate 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-Color-V2, the Yocto-SPI, the YoctoHub-Ethernet or the YoctoHub-GSM-4G

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 a 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

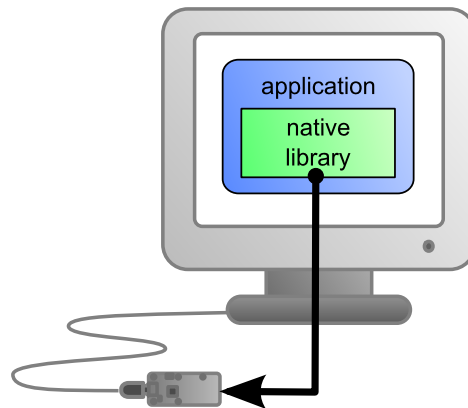
Free space for uploading new files to the filesystem, in bytes.

7.8. What interface: Native, DLL or Service ?

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

Native control

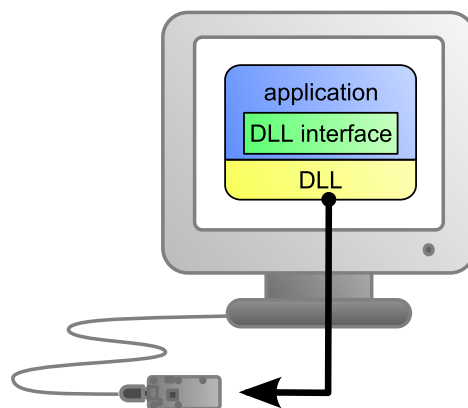
In this case, the software driving your project is compiled directly with a library which provides control of the modules. Objectively, it is the simplest and most elegant solution for the end user. The end user then only needs to plug the USB cable and run your software for everything to work. Unfortunately,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



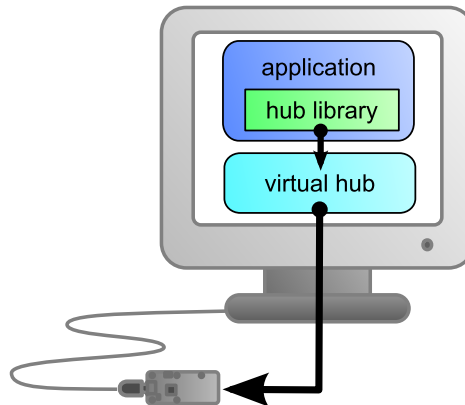
The application uses the DLL to natively control the locally connected module

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 VirtualHub before running the project

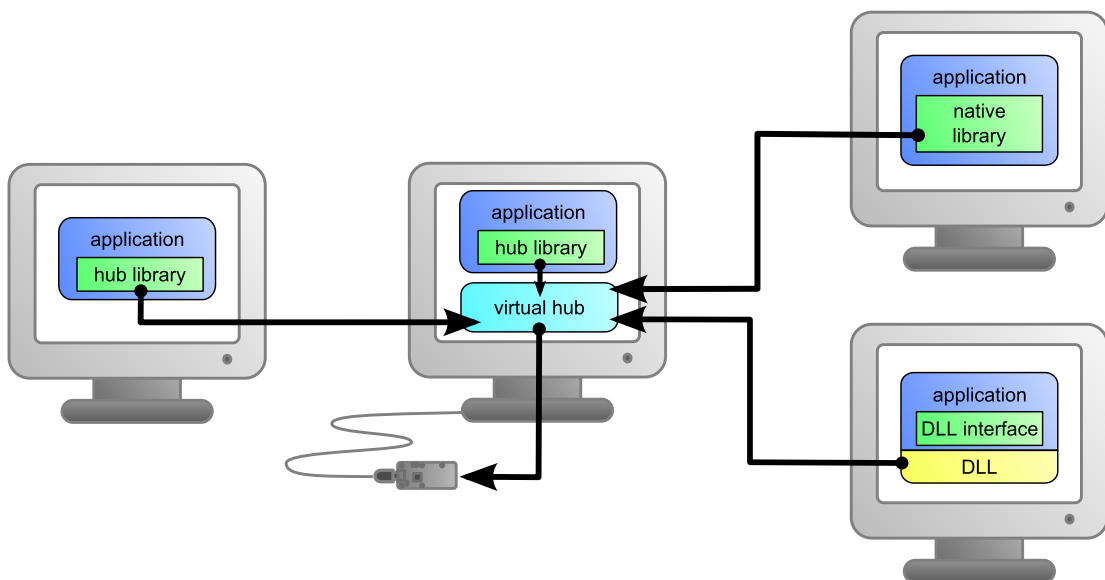
¹ www.yoctopuce.com/EN/virtualhub.php

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



The application connects itself to 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 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
Command line	✓	-	✓
Python	-	✓	✓
C++	✓	✓	✓
C# .Net	-	✓	✓
C# UWP	✓	-	✓
LabVIEW	-	✓	✓
Java	-	✓	✓
Java for Android	✓	-	✓
TypeScript	-	-	✓
JavaScript / ECMAScript	-	-	✓
PHP	-	-	✓
VisualBasic .Net	-	✓	✓
Delphi	-	✓	✓
Objective-C	✓	-	✓

Support methods for different languages

7.9. Accessing modules through a hub

VirtualHub to work around USB access limitation

Only one application at a given time can have native access to Yoctopuce devices. This limitation is related to the fact that two different processes cannot talk to a USB device at the same time. Usually, this kind of problem is solved by a driver that takes care of the police work to prevent multiple processes fighting over the same device. But Yoctopuce products do not use drivers. Therefore, the first process that manages to access the native mode keeps it for itself until `UnregisterHub` or `FreeApi` is called.

If your application tries to communicate in native mode with Yoctopuce devices, but that another application prevents you from accessing them, you receive the following error message:

Another process is already using yAPI

The solution is to use VirtualHub locally on your machine and to use it as a gateway for your applications. In this way, if all your applications use VirtualHub, you do not have conflicts anymore and you can access all your devices all the time.

With a YoctoHub

A YoctoHub behaves itself exactly like a computer running VirtualHub. The only difference between a program using the Yoctopuce API with modules in native USB and the same program with Yoctopuce modules connected to a YoctoHub is located at the level of the `RegisterHub` function call. To use USB modules connected natively, the `RegisterHub` parameter is `usb`. To use modules connected to a YoctoHub, you must simply replace this parameter by the IP address of the YoctoHub.

So there are three possible modes: native mode, network mode via VirtualHub on your local machine, or via a YoctoHub. To switch from native to network mode on your local machine, you only need to change the parameter when calling `YAPI.RegisterHub`, as shown in the examples below:

```
YAPI.RegisterHub("usb",errmsg); // native USB mode

YAPI.RegisterHub("127.0.0.1",errmsg); // local network mode with VirtualHub

YAPI.RegisterHub("192.168.0.10",errmsg); // YoctoHub mode, with 192.168.0.10 as YoctoHub IP address
```

7.10. Programming, where to start?

At this point of the user's guide, you should know the main theoretical points of your Yocto-RS232-C. It is now time to practice. You must download the Yoctopuce library for your favorite programming language from the Yoctopuce web site². Then skip directly to the chapter corresponding to the chosen programming language.

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

When you have mastered the basic programming of your module, you can turn to the chapter on advanced programming that describes some techniques that will help you make the most of your Yocto-RS232-C.

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

8. Using the Yocto-RS232-C in command line

When you want to perform a punctual operation on your Yocto-RS232-C, such as reading a value, assigning a logical name, and so on, you can obviously use VirtualHub, but there is a simpler, faster, and more efficient method: the command line API.

The command line API is a set of executables, one by type of functionality offered by the range of Yoctopuce products. These executables are provided pre-compiled for all the Yoctopuce officially supported platforms/OS. Naturally, the executable sources are also provided¹.

8.1. Installing

Download the command line API². You do not need to run any setup, simply copy the executables corresponding to your platform/OS in a directory of your choice. You may add this directory to your PATH variable to be able to access these executables from anywhere. You are all set, you only need to connect your Yocto-RS232-C, open a shell, and start working by typing for example:

```
C:\>YSerialPort any set_serialMode "9600,8N1"
C:\>YSerialPort any set_protocol "Line"
C:\>YSerialPort any reset
C:\>YSerialPort any writeLine "my line of data"
C:\>YSerialPort any readLine
C:\>YSerialPort any reset
```

To use the command line 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.

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

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

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

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] are logically 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-RS232-C, you need the YSerialPort executable file.

For instance, you can launch:

```
C:\>YSerialPort any set_serialMode "9600,8N1"
C:\>YSerialPort any set_protocol "Line"
C:\>YSerialPort any reset
C:\>YSerialPort any writeLine "my line of data"
C:\>YSerialPort any readLine
C:\>YSerialPort any reset
```

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-RS232-C module with the *RS232M1C-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 RS232M1C-123456.serialPort describe
C:\>YSerialPort RS232M1C-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 RS232M1C-12346 set_logicalName MonPremierModule
C:\>YModule RS232M1C-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 RS232M1C-12346 set_logicalName MonPremierModule
C:\>YModule RS232M1C-12346 saveToFlash
```

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

```
C:\>YModule -s RS232M1C-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 `VirtualHub3` on the concerned machine, and use the executables of the command line API with the `-r` option. For example, if you use:

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

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

³ <http://www.yoctopuce.com/EN/virtualhub.php>

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

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

9. Using the Yocto-RS232-C 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 2.7 and 3.x up to the latest official versions. It works under Windows, macOS, and Linux, Intel as well as ARM. Python interpreters are available on the Python web site¹.

9.1. Source files

The Yoctopuce library classes² for Python that you will use are provided as source files. Copy all the content of the *Sources* directory in the directory of your choice and add this directory to the *PYTHONPATH* environment variable. If you use an IDE to program in Python, refer to its documentation to configure it so that it automatically finds the API source files.

9.2. 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 macOS. 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.

9.3. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS232-C. Here is the skeleton of a Python code snippet 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("RS232M1C-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-RS232-C module with serial number `RS232M1C-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("RS232M1C-123456.serialPort")
serialport = YSerialPort.FindSerialPort("RS232M1C-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.

About python imports

This documentation assumes that you are using the Python library downloaded directly from the Yoctopuce website, but if you are using the library installed with PIP, then you will need to prefix all imports with `yoctopuce..` Meaning all the import examples shown in the documentation, such as:

```
from yocto_api import *
```

need to be converted , when the yoctopuce library was installed by PIP, to:

```
from yoctopuce.yocto_api import *
```

reset

The `reset()` method of the object returned by `YSerialPort.FindSerialPort` clear all internal buffers of the serial port.

readLine

The `readLine()` method returns the next CR/LF terminated line received on the serial port, or an empty string if no more lines are found (this function is non-blocking).

writeLine

The `writeLine()` method puts the specified string in the serial port output buffer, followed by CR/LF.

A real example

Launch Python and open the corresponding sample script provided in the directory **Examples/Doc-GettingStarted-Yocto-RS232-C** 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] + ".serialPort")
    if not serialPort.isOnline():
        sys.exit('Module not connected')
else:
    serialPort = YSerialPort.FirstSerialPort()
    if serialPort is None:
        sys.exit('No module connected (check cable)')

    serialPort.set_serialMode("9600,8N1")
    serialPort.set_protocol("Line")
    serialPort.reset()

while True:
    print("Type line to send, or Ctrl-C to exit:")
    line = input(": ") # use raw_input in python 2.x
    if line == "":
        break
    serialPort.writeLine(line)
    YAPI.Sleep(500)
    line = serialPort.readLine()
    if line != "":
        print("Received: " + line)
YAPI.FreeAPI()
```

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.

```
#!/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()

```

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 `ClassName.STATE_INVALID` value, a `get_currentValue` method returns a `ClassName.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-RS232-C 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, C++ is supported with Microsoft Visual Studio 2017 and more recent versions. Under macOS, we support the XCode versions supported by Apple. And under Linux, we support all GCC version published since 2008. 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++ libraries¹ are integrally provided as source files. A section of the low-level library is written in pure C, but you should not need to interact directly with it: everything was done to ensure the simplest possible interaction from C++. The library is naturally also available as binary files, so that you can link it directly if you prefer.

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

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

10.1. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS232-C. Here is the skeleton of a C++ code snippet 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
YSerialPort *serialport;
```

¹ www.yoctopuce.com/EN/libraries.php

```

serialport = YSerialPort::FindSerialPort("RS232M1C-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-RS232-C.

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-RS232-C module with serial number *RS232M1C-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("RS232M1C-123456.serialPort");
YSerialPort *serialport = YSerialPort::FindSerialPort("RS232M1C-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.

reset

The `reset()` method of the object returned by `yFindSerialPort` clear all internal buffers of the serial port.

readLine

The `readLine()` method returns the next CR/LF terminated line received on the serial port, or an empty string if no more lines are found (this function is non-blocking).

writeLine

The `writeLine()` method puts the specified string in the serial port output buffer, followed by CR/LF.

A real example

Launch your C++ environment and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-RS232-C** 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;
    string target;
    YSerialPort *serialPort;

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

    if (argc > 1) {
        target = (string) argv[1];
        serialPort = YSerialPort::FindSerialPort(target + ".serialPort");
    } else {
        serialPort = YSerialPort::FirstSerialPort();
        if (serialPort == NULL) {
            cerr << "No module connected (check USB cable)" << endl;
            return 1;
        }
    }

    serialPort->set_serialMode("9600,8N1");
    serialPort->set_protocol("Line");
    serialPort->reset();

    string line;
    do {
        YAPI::Sleep(500, errmsg);
        do {
            line = serialPort->readLine();
            if(line != "") {
                cout << "Received: " << line << endl;
            }
        } while(line != "");

        cout << "Type line to send, or Ctrl-C to exit:";
        cin >> line;
        serialPort->writeLine(line);
    } while(line != "");

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

10.2. Control of the module part

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

```
#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;
}

```

}

10.3. Error handling

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

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

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

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

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

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

10.4. Integration variants for the C++ Yoctopuce library

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

Integration in source format (recommended)

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

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

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

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

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

- For Windows: the libraries are added automatically
- For macOS: **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 macOS: **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 macOS 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 macOS: **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++
```

11. Using Yocto-RS232-C 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. C# is supported under Windows Visual Studio 2017 and its more recent versions.

Our programming library is also compatible with *Mono*, the open source version of C# that also works on Linux and macOS. Under Linux, use Mono version 5.20 or more recent. Under macOS, support is limited to 32bit systems, which makes it virtually useless nowadays. You will find on our web site various articles that describe how to configure Mono to use our library.

11.1. Installation

Download the Visual C# Yoctopuce library from the Yoctopuce web site¹. 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.

11.2. 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 modules². The source files in Visual C# manage the high level part of the API. Therefore, you need both this DLL and the .cs files of the `sources` directory to create a project managing Yoctopuce modules.

Configuring a Visual C# project

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

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

¹ www.yoctopuce.com/EN/libraries.php

² 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` directory³. Then, from the **Solution Explorer** window, right click on the DLL, select **Properties** and in the **Properties** panel, set the **Copy to output folder** to **always**. You are now ready to use your Yoctopuce modules from Visual Studio.

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

11.3. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS232-C. Here is the skeleton of a C# code snippet 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("RS232M1C-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-RS232-C module with serial number `RS232M1C-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("RS232M1C-123456.serialPort");
serialport = YSerialPort.FindSerialPort("RS232M1C-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.

³ 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.

reset

The `reset()` method of the object returned by `YSerialPort.FindSerialPort` clear all internal buffers of the serial port.

readLine

The `readLine()` method returns the next CR/LF terminated line received on the serial port, or an empty string if no more lines are found (this function is non-blocking).

writeLine

The `writeLine()` method puts the specified string in the serial port output buffer, followed by CR/LF.

A real example

Launch Microsoft Visual C# and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-RS232-C** 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 errormsg = "";
            string target;
            YSerialPort serialPort;

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

            if (args.Length > 0 && args[0] != "any") {
                target = args[0];
                serialPort = YSerialPort.FindSerialPort(target + ".serialPort");
                if (!serialPort.isOnline()) {
                    Console.WriteLine("No module connected (check cable)");
                    Environment.Exit(0);
                }
            } else {
                serialPort = YSerialPort.FirstSerialPort();
                if (serialPort == null) {
                    Console.WriteLine("No module connected (check USB cable)");
                    Environment.Exit(0);
                }
            }

            serialPort.set_serialMode("9600,8N1");
            serialPort.set_protocol("Line");
            serialPort.reset();

            string line;
            do {
                YAPI.Sleep(500, ref errormsg);
                do {
                    line = serialPort.readLine();
                    if (line != "") {

```

```

        Console.WriteLine("Received: " + line);
    }
    } while (line != "");
    Console.WriteLine("Type line to send, or Ctrl-C to exit: ");
    line = Console.ReadLine();
    serialPort.WriteLine(line);
    } while (line != "");
    YAPI.FreeAPI();
}
}
}

```

11.4. Control of the module part

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

```

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

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

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

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

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

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

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

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

```

```

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

```

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

Changing the module settings

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

```

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

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

        static void Main(string[] args)
        {
            YModule m;
            string 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.WriteLine("Module: serial= " + m.get_serialNumber());
                Console.WriteLine(" / name= " + m.get_logicalName());
            } else {
                Console.WriteLine("not connected (check identification and USB cable)");
            }
            YAPI.FreeAPI();
        }
    }
}

```

```
}
```

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

Listing the modules

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

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            YModule m;
            string 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();
        }
    }
}
```

11.5. Error handling

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

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

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

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

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

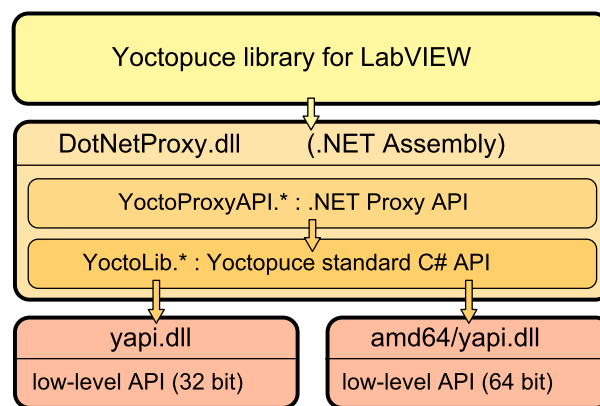
When you work without exceptions, you can obtain an error code and an error message explaining the source of the error. You can request them from the object which returned the error, calling the `errType()` and `errMessage()` methods. Their returned values contain the same information as in the exceptions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

12. Using the Yocto-RS232-C 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.

12.1. 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 of 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.

12.2. 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 DotNetProxyLibrary.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.

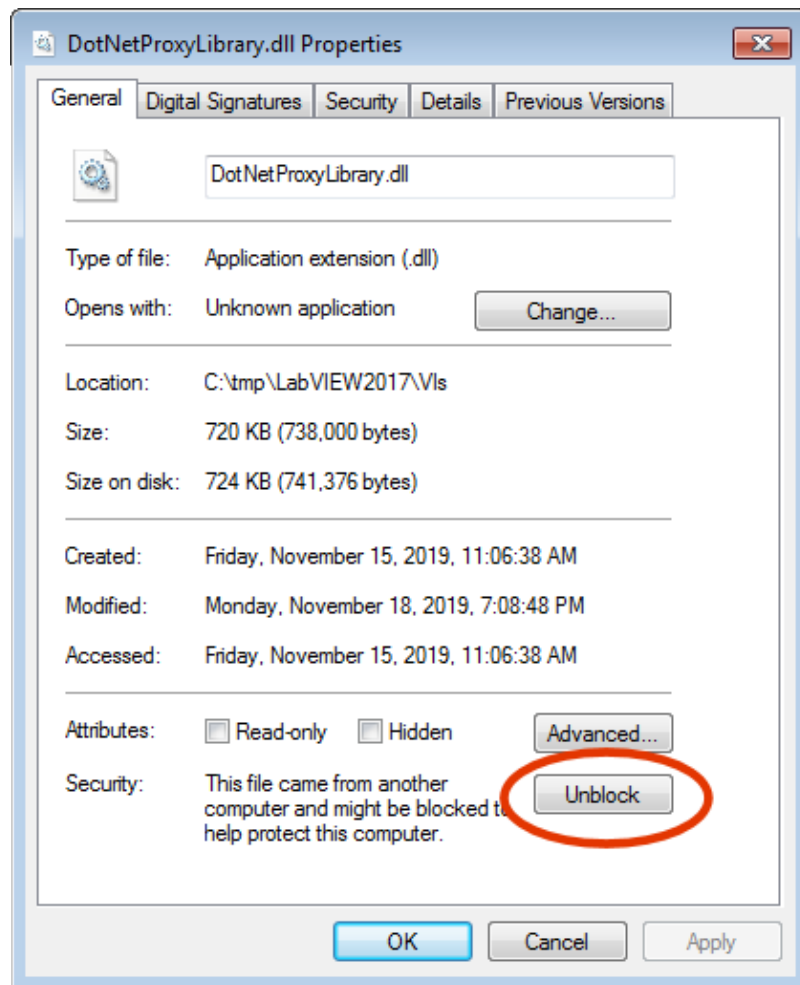
12.3. Installation

Download the LabVIEW library from the Yoctopuce web site¹. 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

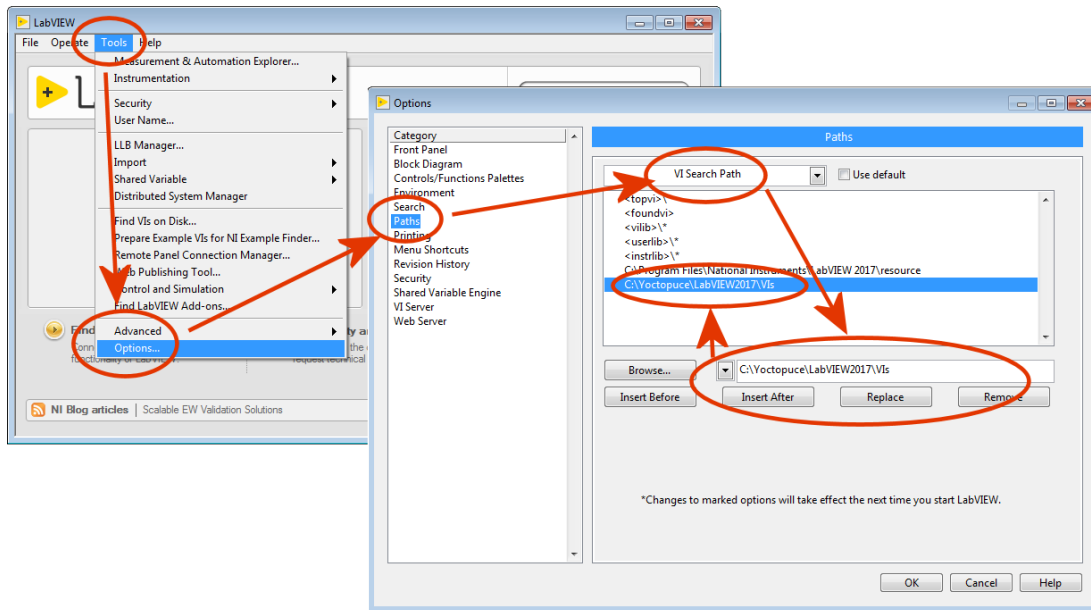
To install the LabVIEW Yoctopuce API, there are several methods.

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.

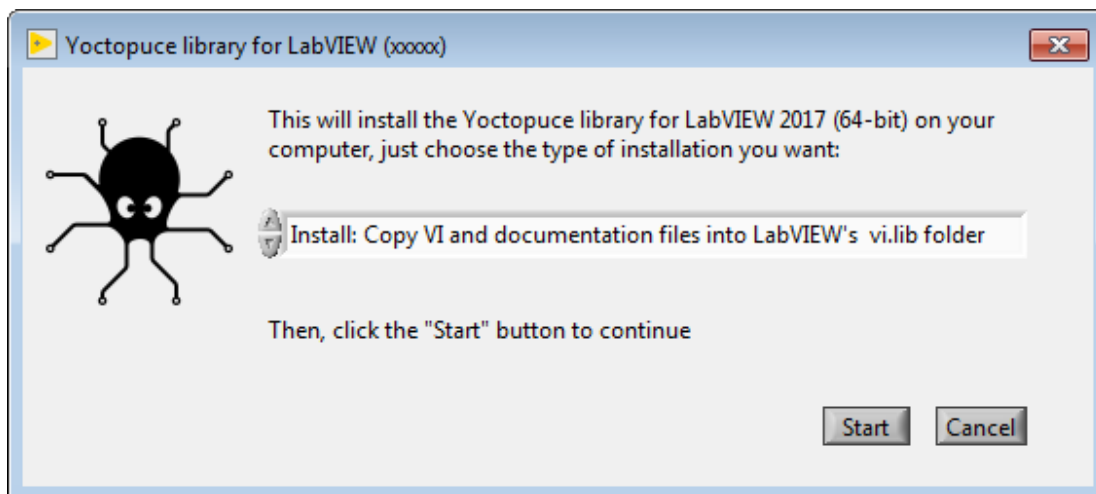
² <https://knowledge.ni.com/KnowledgeArticleDetails?id=kA00Z000000P8XnSAK>



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.

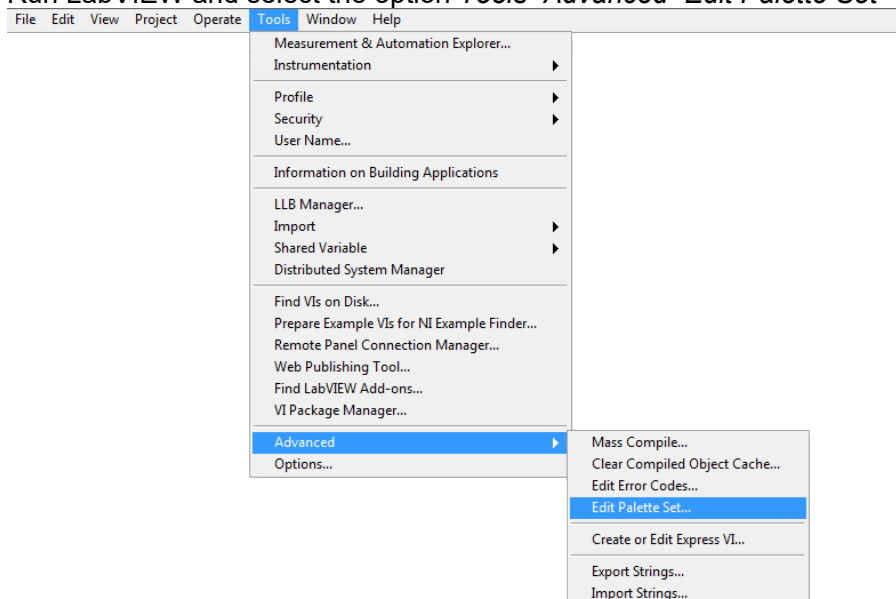
The installer identifies Yoctopuce VIs library folders by checking the presence of the *YRegisterHub.vi* file in said folders.

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:

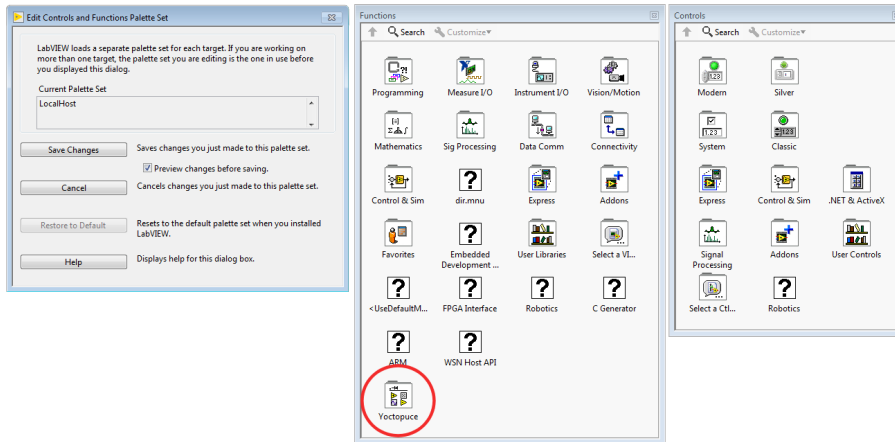
1. Create a *Yoctopuce/API* directory in the *C:\Program Files\National Instruments\LabVIEW xxxx\vi.lib* directory and copy all the VIs and DLLs of the *VIs* directory into it.
2. Create a *Yoctopuce* directory in the *C:\Program Files\National Instruments\LabVIEW xxxx\menus\Categories* directory.
3. 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>

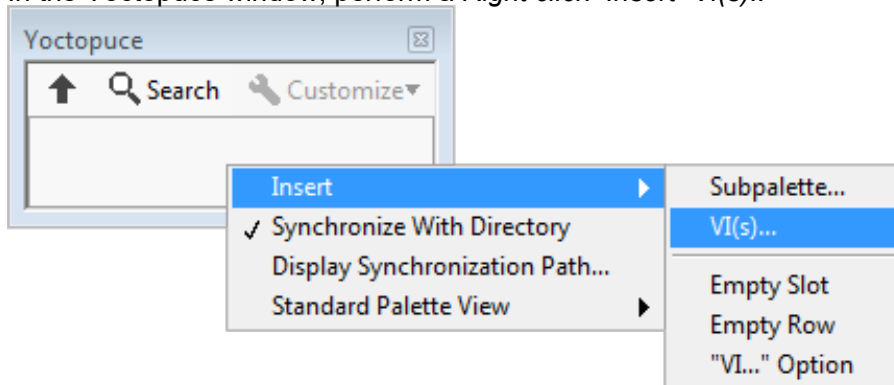
Three windows pop up:

- "Edit Controls and Functions Palette Set"
- "Functions"
- "Controls"

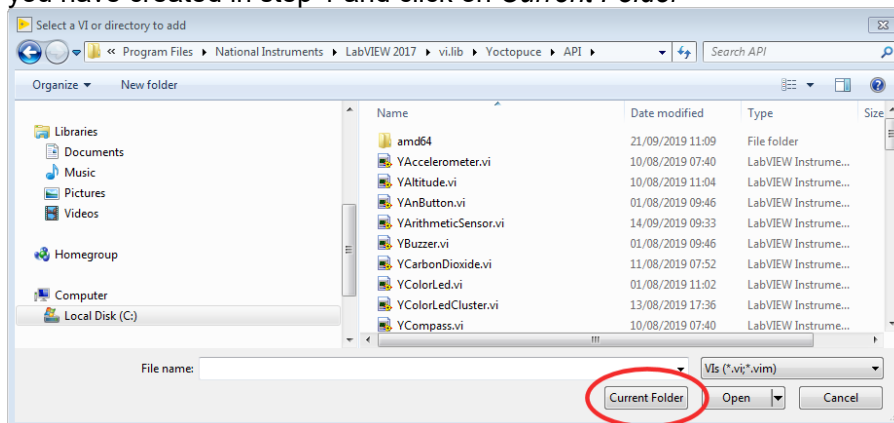


In the *Function* window, there is a *Yoctopuce* icon. Double-click it to create an empty "Yoctopuce" window.

4. In the Yoctopuce window, perform a *Right click>Insert>Vi(s)..*

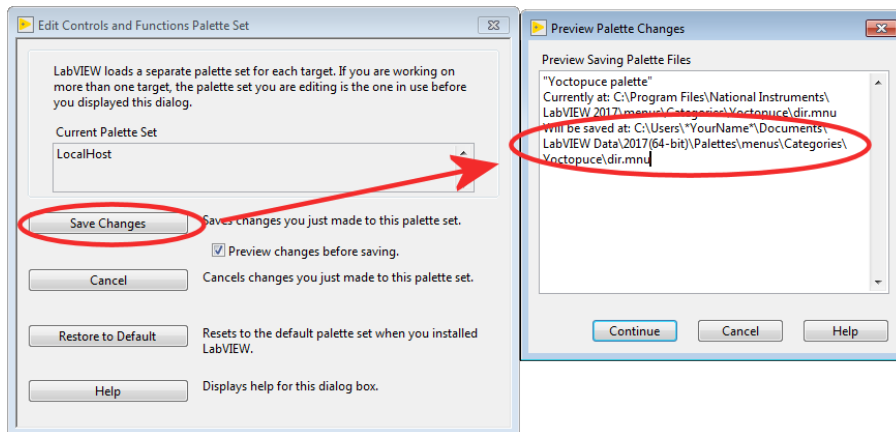


in order to open a file chooser. Put the file chooser in the *vi.lib\Yoctopuce\API* directory that you have created in step 1 and click on *Current Folder*



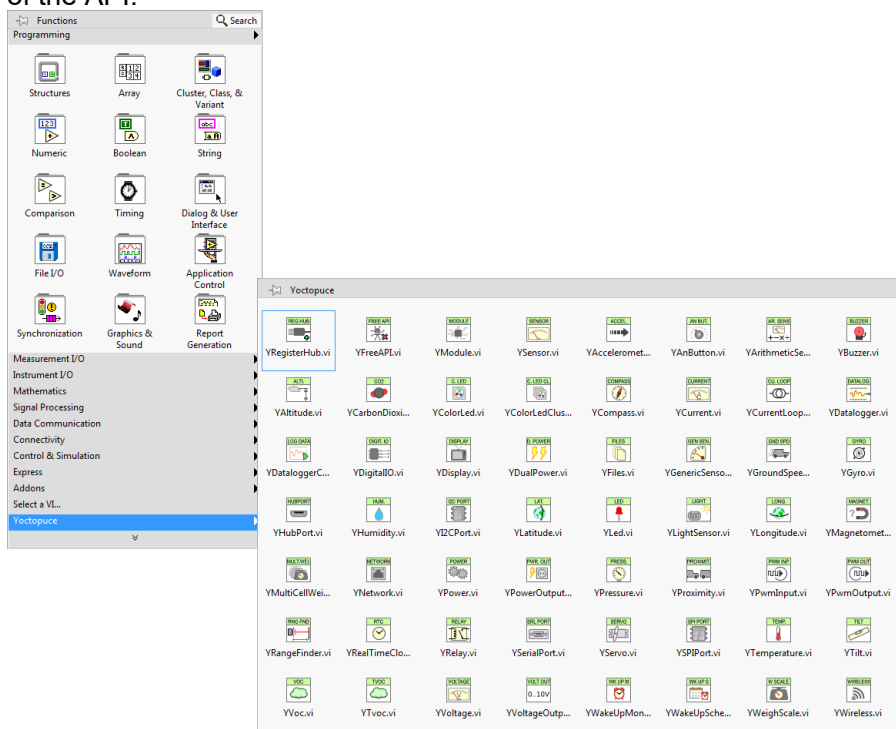
All the Yoctopuce VIs now appear in the Yoctopuce window. By default, they are sorted by alphabetical order, but you can arrange them as you see fit by moving them around with the mouse. For the palette to be easy to use, we recommend to reorganize the icons over 8 columns.

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.

- Restart LabVIEW, the LabVIEW palette now contains a Yoctopuce sub-palette with all the VIs of the API.

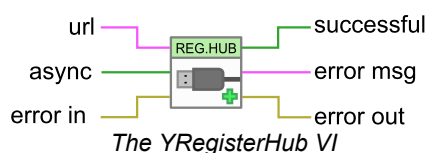


12.4. Presentation of Yoctopuce VIs

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 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 YoctoHub⁴ or even that of a machine on which the VirtualHub⁵ application is running.

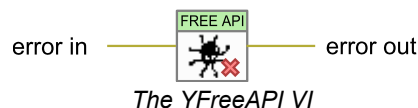
In the case of an IP address, the YRegisterHub VI tries to contact this address and generates an error if it does not succeed, unless the *async* parameter is set to TRUE. If *async* is set to TRUE, no error is generated and Yoctopuce modules corresponding to that IP address become automatically available 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

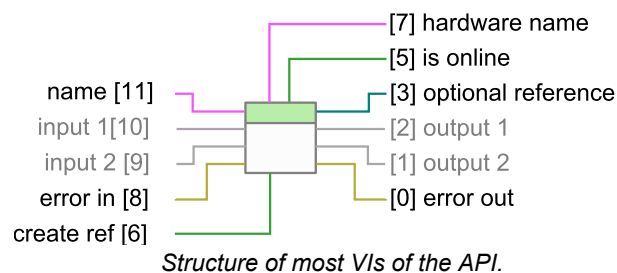
The YFreeAPI VI enables you to free resources allocated by the Yoctopuce API.



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:



- 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 VI⁶. 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.

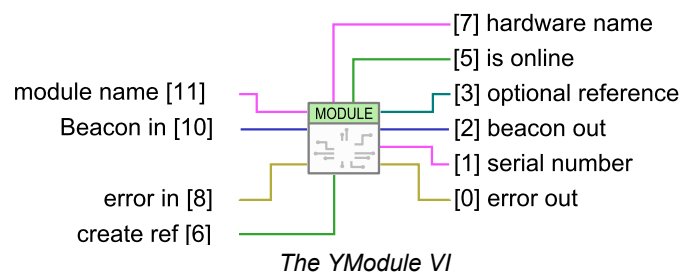
You can find the list of functions available on your Yocto-RS232-C in chapter *Programming, general concepts*.

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.

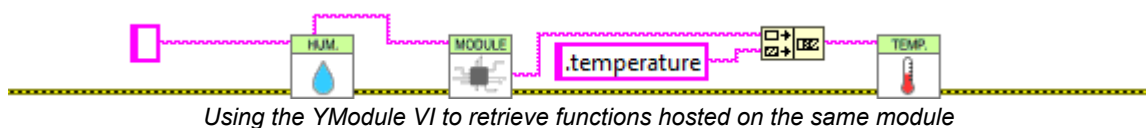
If the *name* parameter contains an empty string, the VI targets the first available function of the same type. If no function is available, *is online* is set to FALSE.

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.

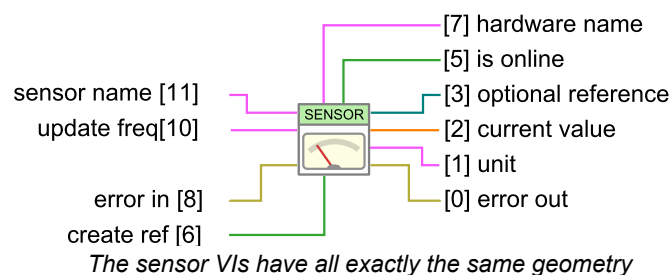


The *name* input works slightly differently from other VIs. If it is called with a *name* parameter corresponding to a function name, the `YModule` VI finds the *Module* function of the module hosting the function. You can therefore easily find the serial number of the module of any function. This enables 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.



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.

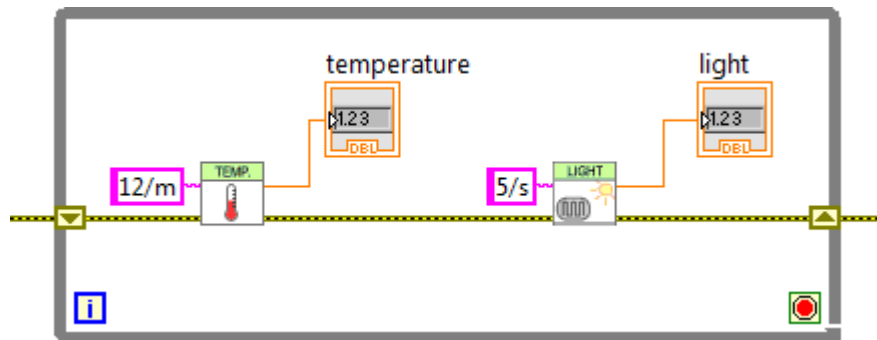


The *update freq* input parameter is a character string enabling you to configure the way in which the output value is updated:

- "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.

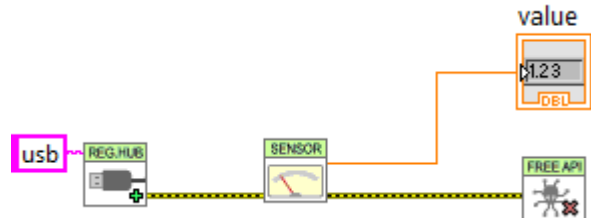


Changing the update frequency of the same module

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.

12.5. Functioning and use of VIs

Here is one of the simplest example of VIs using the Yoctopuce API.

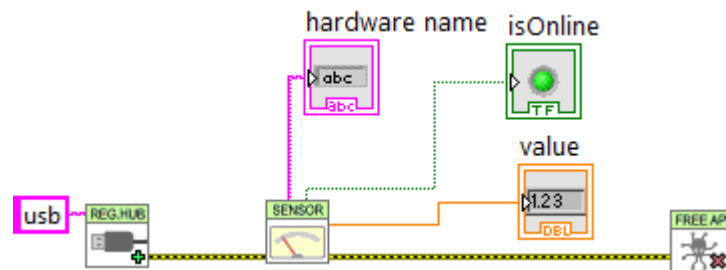


Minimal example of use of the LabVIEW Yoctopuce API

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:

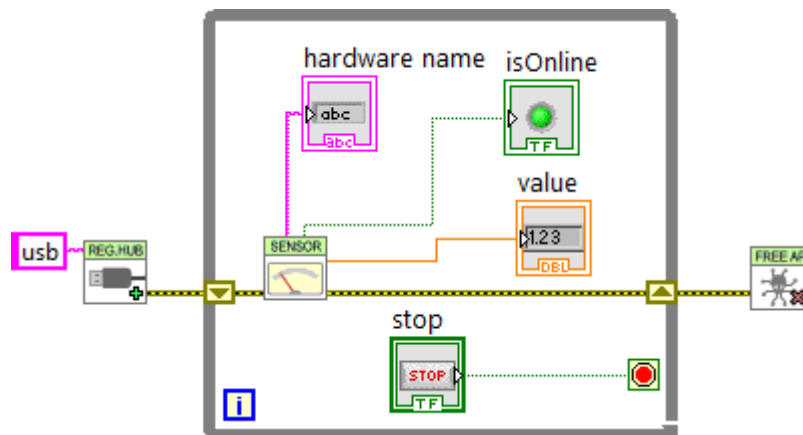
1. It initializes the API in native ("usb") mode with the `YRegisterHub` VI.
2. It displays the value of the first Yoctopuce sensor it finds thanks to the `YSensor` VI.
3. It frees the API thanks to the `YFreeAPI` VI.

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

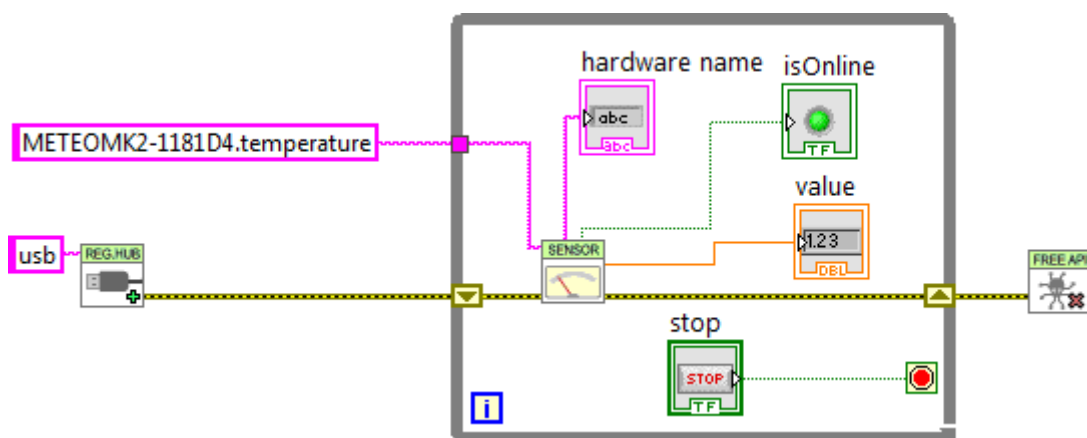
The VIs of the Yoctopuce API are actually an entry door into the library. Internally, this mechanism works independently of the Yoctopuce VIs. Indeed, most communications with electronic modules are managed automatically as background tasks. Therefore, you do not necessarily need to take any specific care to use Yoctopuce VIs, you can for example use them in a non-delayed loop without creating any specific problem for the API.



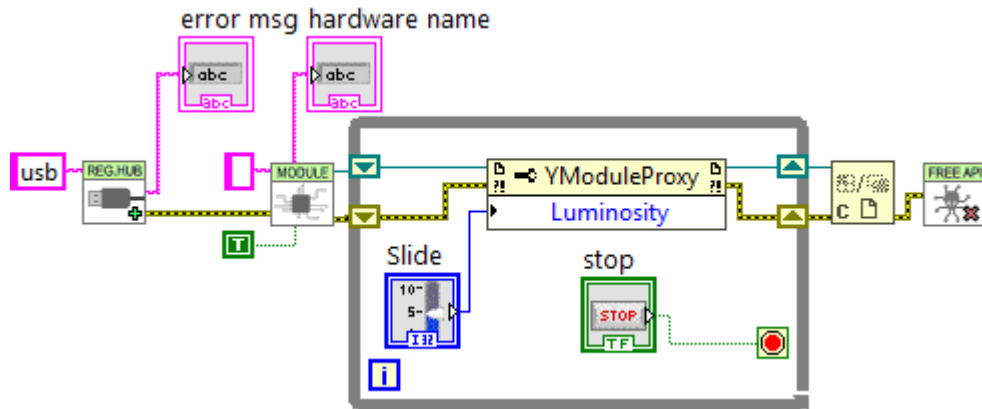
The Yoctopuce VIs can be used in a non-delayed loop

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.

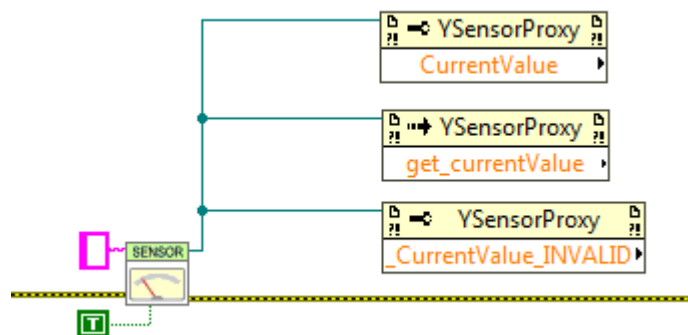


Using names to identify the functions to be used



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.

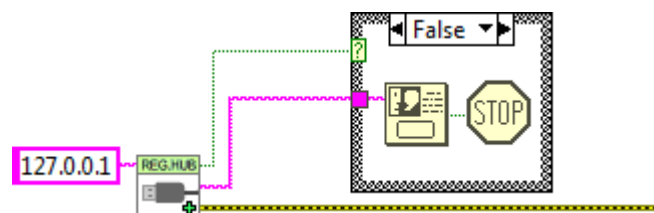


Property and Invoke nodes: Using properties, methods and constants

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 YoctoHubs⁷ or to 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.

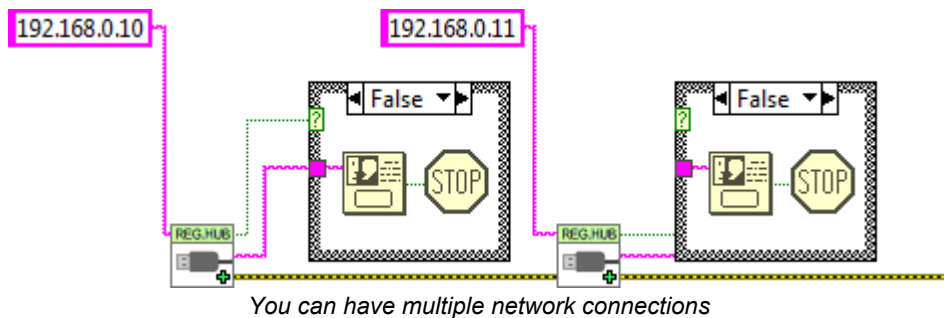


Network mode

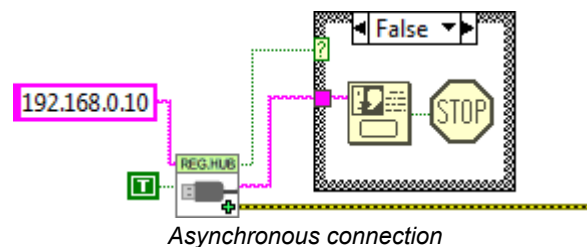
⁷ <https://www.yoctopuce.com/EN/products/category/extensions-and-networking>

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.



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.

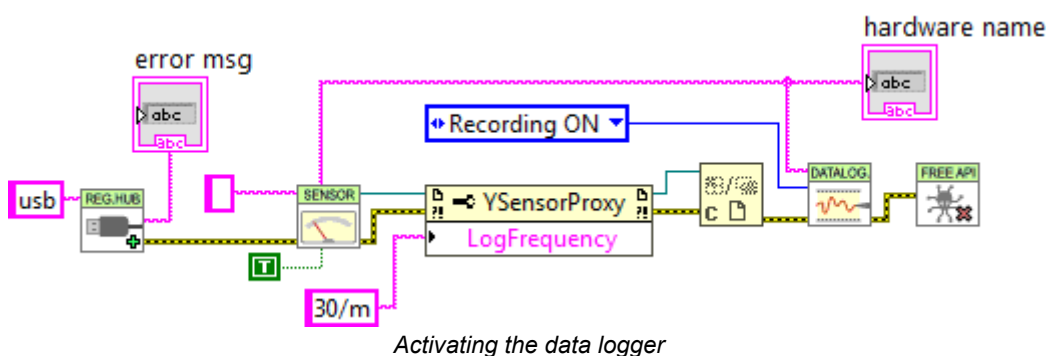


12.7. Managing the data logger

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.

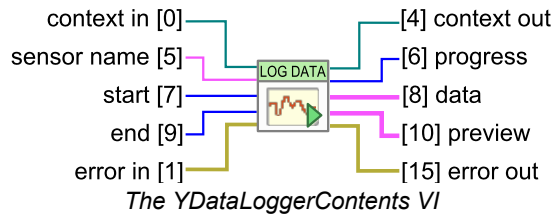
Logging

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.



Reading

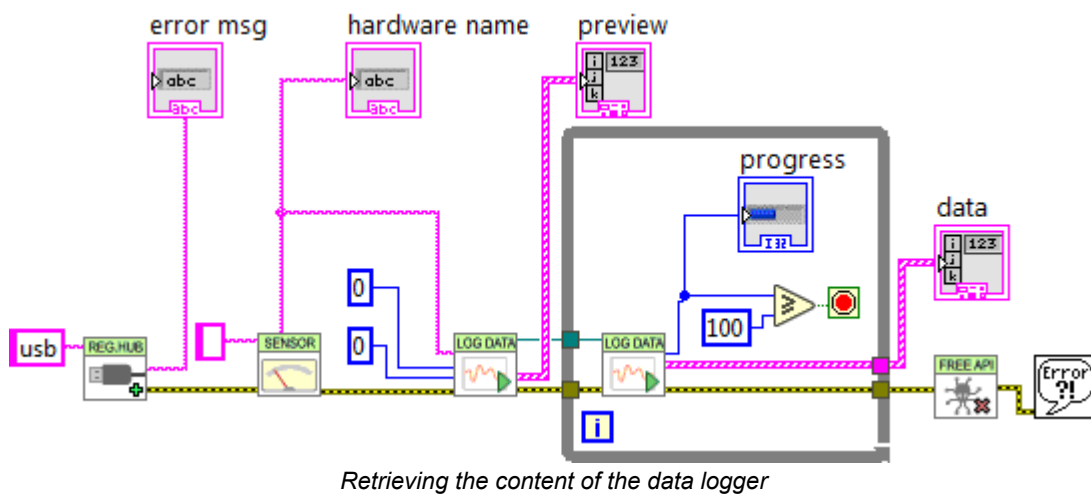
You can retrieve the data in the data logger with the YDataLoggerContents VI.



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.



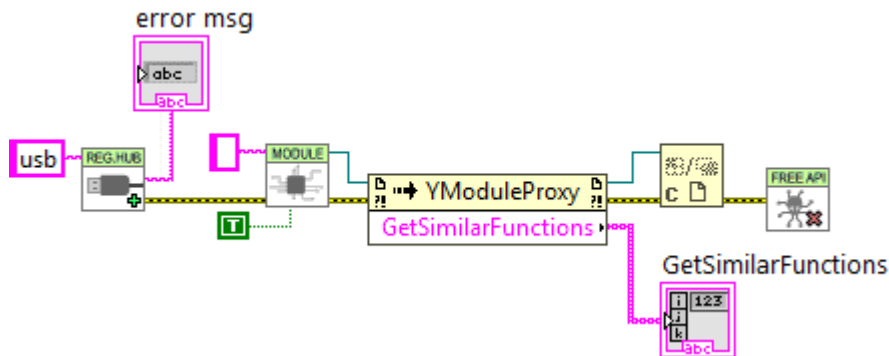
The results and the summary are returned as an array of structures containing the following fields:

- *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.

12.8. Function list

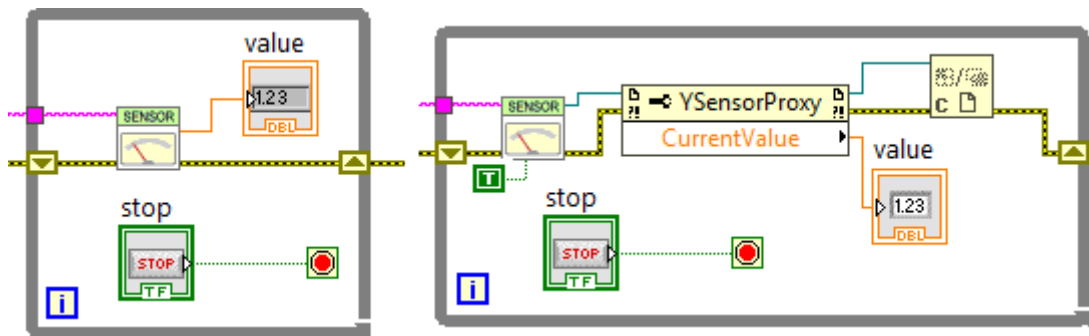
Each VI corresponding to an object of the *Proxy API* enables you to list all the functions of the same class with the *getSimilarFunctions()* method of the corresponding *Proxy* object. Thus, you can easily perform an inventory of all the connected modules, of all the connected sensors, of all the connected relays, and so on.



Retrieving the list of all the modules which are connected

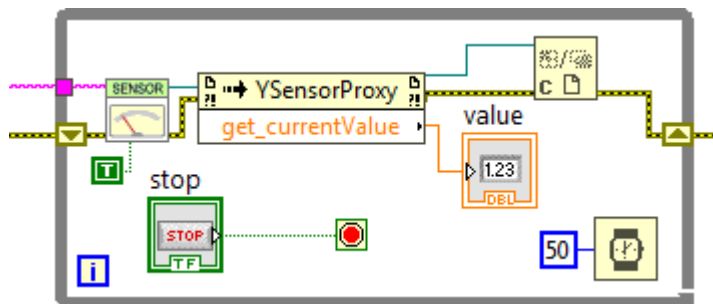
12.9. A word on performances

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

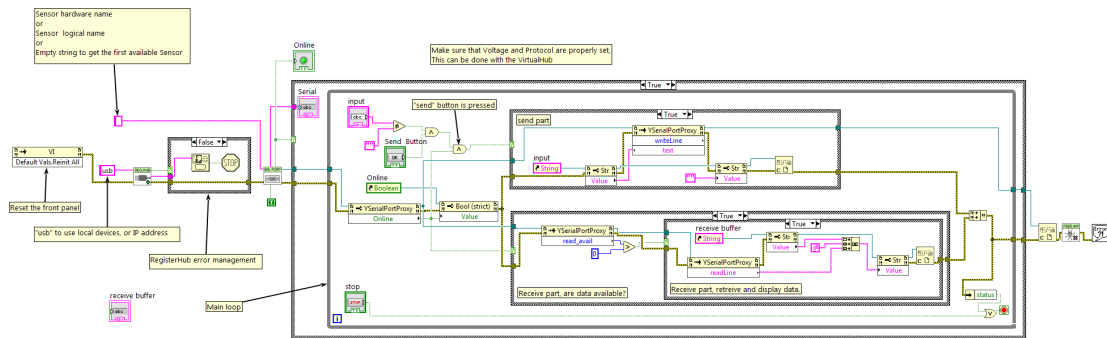
However, almost all the methods of the available Proxy objects initiate a communication with the Yoctopuce modules each time they are called. You should therefore avoid calling them too often without purpose.



This loop, using a method, must be slowed down

12.10. A full example of a LabVIEW program

Here is an example using a Yocto-RS232-C 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 send the contents of the input field as soon as the "send" button is pressed. The transmission is achieved through the *YSerialPortProxy*'s "writeLine" method. In parallel, the application checks if there is anything available in the reception buffer, and display it if so. Note how all references are closed once they are not needed anymore. When the application is about to close, the Yoctopuce API is closed thanks to the *YFreeAPI* VI.



Example of Yocto-RS232-C usage in LabVIEW

If you read this documentation on screen, you can zoom on the image above. You can also find this example in the LabVIEW Yoctopuce library.

12.11. 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.

13. Using the Yocto-RS232-C 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 VirtualHub to communicate with Yoctopuce devices.

13.1. Getting ready

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

- The Java programming library¹
- VirtualHub² for Windows, macOS 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 VirtualHub, 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.

13.2. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS232-C. 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("RS232M1C-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-RS232-C module with serial number *RS232M1C-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("RS232M1C-123456.serialPort")
serialport = YSerialPort.FindSerialPort("RS232M1C-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.

reset

The `reset()` method of the object returned by `YSerialPort.FindSerialPort` clear all internal buffers of the serial port.

readLine

The `readLine()` method returns the next CR/LF terminated line received on the serial port, or an empty string if no more lines are found (this function is non-blocking).

writeLine

The `writeLine()` method puts the specified string in the serial port output buffer, followed by CR/LF.

A real example

Launch your Java environment and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-RS232-C** 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.YAPI;
import com.yoctopuce.YoctoAPI.YAPI_Exception;
import com.yoctopuce.YoctoAPI.YSerialPort;

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) {
            String target = args[0];
            serialPort = YSerialPort.FindSerialPort(target + ".serialPort");
        } else {
            serialPort = YSerialPort.FirstSerialPort();
            if (serialPort == null) {
                System.out.println("No module connected (check USB cable)");
                System.exit(1);
            }
        }

        InputStreamReader inputStreamReader = new InputStreamReader(System.in);
        BufferedReader console = new BufferedReader(inputStreamReader);
        try {
            serialPort.set_serialMode("9600,8N1");
            serialPort.set_protocol("Line");
            serialPort.reset();
            String line;
            do {
                YAPI.Sleep(500);
                do {
                    line = serialPort.readLine();
                    if (!line.equals("")) {
                        System.out.println("Received: " + line);
                    }
                } while (!line.equals(""));
                System.out.println("Type line to send, or Ctrl-C to exit:");
                line = console.readLine();
                serialPort.writeLine(line);
            } while (!line.equals(""));
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        YAPI.FreeAPI();
    }
}

```

13.3. Control of the module part

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

```

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();
    }
}

```

```
}
```

13.4. Error handling

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

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

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.

14. Using the Yocto-RS232-C 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.

14.1. 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 VirtualHub running on another OS.

14.2. Getting ready

Go to the Yoctopuce web site and download the Java for Android programming library¹. The library is available as source files, and also as a jar file. Connect your modules, decompress the library files in the directory of your choice, and configure your Android programming environment so that it can find them.

To keep them simple, all the examples provided in this documentation are snippets of Android applications. You must integrate them in your own Android applications to make them work. However, you can find complete applications in the examples provided with the Java for Android library.

14.3. 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.

Android version

Our library can be compiled to work with older versions, as long as the Android tools allow us to support them, i.e. approximately versions of the last ten years.

¹ www.yoctopuce.com/EN/libraries.php

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.

14.4. 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 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>
```

14.5. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS232-C. 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("RS232M1C-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-RS232-C module with serial number *RS232M1C-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("RS232M1C-123456.serialPort")
serialport = YSerialPort.FindSerialPort("RS232M1C-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.

reset

The `reset()` method of the object returned by `YSerialPort.FindSerialPort` clear all internal buffers of the serial port.

readLine

The `readLine()` method returns the next CR/LF terminated line received on the serial port, or an empty string if no more lines are found (this function is non-blocking).

writeLine

The `writeLine()` method puts the specified string in the serial port output buffer, followed by CR/LF.

A real example

Launch your Java environment and open the corresponding sample project provided in the directory **Examples//Doc-Examples** of the Yoctopuce library.

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

```
package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ArrayAdapter;
import android.widget.EditText;
import android.widget.SeekBar;
import android.widget.Spinner;
import android.widget.TextView;

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

```

import com.yoctopuce.YoctoAPI.YPwmOutput;
import com.yoctopuce.YoctoAPI.YSerialPort;

public class GettingStarted_Yocto_RS232 extends Activity implements OnItemSelectedListener
{

    private YSerialPort serialPort = null;
    private ArrayAdapter<String> aa;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.gettingstarted_yocto_rs232);
        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();
        aa.clear();
        try {
            YAPI.EnableUSBHost(this);
            YAPI.RegisterHub("usb");
            YSerialPort s = YSerialPort.FirstSerialPort();
            while (s != null) {
                String hwid = s.get_hardwareId();
                aa.add(hwid);
                s = s.nextSerialPort();
            }
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
        aa.notifyDataSetChanged();
    }

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

    @Override
    public void onItemSelected(AdapterView<?> parent, View view, int pos, long id)
    {
        String hwid = (String) parent.getItemAtPosition(pos);
        serialPort = YSerialPort.FindSerialPort(hwid);
        try {
            serialPort.set_serialMode("9600,8N1");
            serialPort.set_protocol("Line");
            serialPort.reset();
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
    }

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

    /** Called when the user touches the button State A */
    public void update(View view)
    {
        if (serialPort == null)
            return;

        try {

            EditText editText = (EditText) findViewById(R.id.editText1);
            String line = editText.getText().toString();
            serialPort.writeLine(line);

```

```

        YAPI.Sleep(500);
        String response = "";
        String outline;
        do {
            outline = serialPort.readLine();
            response += outline;
        } while (!outline.equals(""));
        TextView textView = (TextView) findViewById(R.id.response);
        textView.setText(response);
    } catch (YAPI_Exception e) {
        e.printStackTrace();
    }
}
}

```

14.6. Control of the module part

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

```

package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ArrayAdapter;
import android.widget.Spinner;
import android.widget.Switch;
import android.widget.TextView;

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

public class ModuleControl extends Activity implements OnItemClickListener
{
    private ArrayAdapter<String> aa;
    private YModule module = null;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.modulecontrol);
        Spinner my_spin = (Spinner) findViewById(R.id.spinner1);
        my_spin.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 onItemClick(AdapterView<?> parent, View view, int pos, long id)
    {
        String hwid = parent.getItemAtPosition(pos).toString();
        module = YModule.FindModule(hwid);
        DisplayModuleInfo();
    }

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

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

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

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

```

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

Changing the module settings

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

```
package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ArrayAdapter;
import android.widget.EditText;
import android.widget.Spinner;
import android.widget.TextView;
import android.widget.Toast;

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

public class SaveSettings extends Activity implements OnItemClickListener
{
    private ArrayAdapter<String> aa;
    private YModule module = null;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.savesettings);
        Spinner my_spin = (Spinner) findViewById(R.id.spinner1);
        my_spin.setOnItemClickListener(this);
        aa = new ArrayAdapter<String>(this, android.R.layout.simple_spinner_item);
        aa.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
        my_spin.setAdapter(aa);
    }

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

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

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

    private void DisplayModuleInfo()
    {

```

```

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

    @Override
    public void 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();
    }
}

```

14.7. 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.

15. Using Yocto-RS232-C with TypeScript

TypeScript is an enhanced version of the JavaScript programming language. It is a syntactic superset with strong typing, therefore increasing the code reliability, but transpiled - aka compiled - into JavaScript for execution in any standard Web browser or Node.js environment.

This Yoctopuce library therefore makes it possible to implement JavaScript applications using strong typing. Similarly to our EcmaScript library, it uses the new asynchronous features introduced in ECMAScript 2017, which are now available in all modern JavaScript environments. Note however that at the time of writing, Web browsers and Node.JS cannot use TypeScript code directly, so you must first compile your TypeScript into JavaScript before running it.

The library works both in a Web browser and in Node.js. In order to allow for a static resolution of dependencies, and to avoid ambiguities that can arise when using hybrid environments such as Electron, the choice of the runtime environment must be done explicitly upon import of the library, by referencing in the project either `yocto_api_nodejs.js` or `yocto_api_html.js`.

The library can be integrated in your projects in multiple ways, depending on what best fits your requirements:

- by directly copying the TypeScript library source files into your project, and by adding them to your build script. Only a few files are usually needed to handle most use-cases. You will find TypeScript source files in the `src` subdirectory of our library.
- by using CommonJS module resolution, natively supported by TypeScript, with a package manager such as `npm`. You will find a version of the library transpiled according to CommonJS module standard in the `dist/cjs` subdirectory, including all type definition files (with extension `.d.ts`) and source maps (with extension `.js.map`) enabling source-level error reporting and debugging. We have also published these files on `npmjs` under the name `yoctolib-cjs`.
- by using ECMAScript standard module resolution, also supported by TypeScript, usually referenced by relative path. You will find a version of the library transpiled as an ECMAScript 2015 module in the `dist/esm` subdirectory, including all type definition files (with extension `.d.ts`) and source maps (with extension `.js.map`) enabling source-level error reporting and debugging. We have also published these files on `npmjs` under the name `yoctolib-esm`.

15.1. Using the Yoctopuce library for TypeScript

1. Start by installing TypeScript on your machine if this is not yet done. In order to do so:

- Install on your development machine the official version of Node.js (version 10 or more recent). You can download it for free from the official web site: <http://nodejs.org>. Make sure to install it fully, including `npm`, and add it to the system path.
- Then install TypeScript on your machine using the command line:

```
npm install -g typescript
```

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

- The TypeScript programming library¹
- The VirtualHub software² for Windows, macOS, or Linux, depending on your OS. TypeScript and JavaScript are part of those languages which do not generally allow you to directly access to USB peripherals. 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*. No extra driver will be needed, though.

3. Extract the library files in a folder of your choice, and open a command window in the directory where you have installed it. In order to install the few dependencies which are necessary to start the examples, run this command:

```
npm install
```

When the command has run without error, you are ready to explore the examples. They are available in two different trees, depending on the environment that you need to use: `example_html` for running the Yoctopuce library within a Web browser, or `example_nodejs` if you plan to use the library in a Node.js environment.

The method to use for launching the examples depends on the environment. You will find more about it below.

15.2. Refresher on asynchronous I/O in JavaScript

JavaScript is single-threaded by design. In order to handle time-consuming I/O operations, JavaScript relies on asynchronous operations: the I/O call is only triggered but then the code execution flow is suspended. The JavaScript engine is therefore free to handle other pending tasks, such as user interface. 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, the ECMAScript 2015 standard came in with *Promise* objects and a new `async / await` syntax to abstract calls to asynchronous calls:

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

To make a long story short, *async* and *await* make it possible to write TypeScript code with all the benefits of asynchronous I/O, but without breaking the code flow. It is almost like multi-threaded

¹ www.yoctopuce.com/EN/libraries.php

² www.yoctopuce.com/EN/virtualhub.php

execution, except that control switch between pending tasks only happens at places where the *await* keyword appears.

This TypeScript library uses the *Promise* objects and *async* methods, to allow you to use the *await* syntax. To keep it easy to remember, all public methods of the TypeScript 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 works on internal device lists handled in background, and does not require immediate asynchronous I/O.

In most cases, TypeScript strong typing will remind you to use *await* when calling an asynchronous method.

15.3. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS232-C. Here is the skeleton of a TypeScript code snippet to use the SerialPort function.

```
// For Node.js, the library is referenced through the NPM package
// For HTML, we would use instead a relative path (depending on the build environment)
import { YAPI, YErrorMsg, YModule } from 'yoctolib-cjs/yocto_api_nodejs.js';
import { YSerialPort } from 'yoctolib-cjs/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 = YSerialPort.FindSerialPort("RS232M1C-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 imports 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-RS232-C. Other imports can be useful in other cases, such as `YModule` which can let you enumerate any type of Yoctopuce device.

In order to properly bind `yocto_api` to the proper network libraries (provided either by Node.js or by the web browser for an HTML application), you must import at least once in your project one of the two variants `yocto_api_nodejs.js` or `yocto_api_html.js`.

Note that this example imports the Yoctopuce library as a CommonJS module, which is the most frequently used with Node.JS, but if your project is designed around EcmaScript native modules, you can also replace in the import directive the prefix `yoctolib-cjs` by `yoctolib-esm`.

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.

As explained above, using `RegisterHub("usb")` is not supported in TypeScript, because the JavaScript engine has no direct access to USB devices. It needs to go through the VirtualHub via a localhost connection.

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-RS232-C module with serial number *RS232M1C-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("RS232M1C-123456.serialPort")
serialport = YSerialPort.FindSerialPort("RS232M1C-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.

reset

The `reset()` method of the object returned by `YSerialPort.FindSerialPort` clear all internal buffers of the serial port.

readLine

The `readLine()` method returns the next CR/LF terminated line received on the serial port, or an empty string if no more lines are found (this function is non-blocking).

writeLine

The `writeLine()` method puts the specified string in the serial port output buffer, followed by CR/LF.

A real example, for Node.js

Open a command window (a terminal, a shell...) and go into the directory **example_nodejs/Doc-GettingStarted-Yocto-RS232-C** within Yoctopuce library for TypeScript. In there, you will find a file named `demo.ts` 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-RS232-C 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-RS232-C is connected and where you run the VirtualHub.

```
import { YAPI, YErrorMsg, YModule } from 'yoctolib-cjs/yocto_api_nodejs.js';
import { YSerialPort } from 'yoctolib-cjs/yocto_serialport.js';
import * as readline from 'readline';

let serialPort: YSerialPort;

async function startDemo(): Promise<void>
{
    await YAPI.LogUnhandledPromiseRejections();

    // Setup the API to use the VirtualHub on local machine
    let errmsg: YErrorMsg = 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: string = process.argv[process.argv.length-1];
  if (serial[8] != '-') {
    // by default use any connected module suitable for the demo
    let anysensor = YSerialPort.FirstSerialPort();
    if (anysensor) {
      let module: YModule = await anysensor.get_module();
      serial = await module.get_serialNumber();
    } else {
      console.log('No matching module connected, check cable !');
      await YAPI.FreeAPI();
      return;
    }
  }
  console.log('Using device ' + serial);
  serialPort = YSerialPort.FindSerialPort(serial + ".serialPort");

  await serialPort.set_serialMode("9600,8N1");
  await serialPort.set_protocol("Line");
  await serialPort.reset();

  const rl: readline.Interface = readline.createInterface({
    input: process.stdin,
    output: process.stdout
  });
  console.log('Type line to send, or Ctrl-C to exit:');
  rl.on('line', (input) => {
    serialPort.writeLine(input);
  });

  refresh();
}

async function refresh(): Promise<void>
{
  let line;
  if (await serialPort.isOnline()) {
    do {
      line = await serialPort.readLine();
      if (line != "") {
        console.log("Received: " + line);
      }
    } while (line != "");
  } else {
    console.log('Module not connected');
  }
  setTimeout(refresh, 500);
}

startDemo();

```

As explained at the beginning of this chapter, you need to have installed the TypeScript compiler on your machine to test these examples, and to install the typescript library dependencies. If you have done that, you can now type the following two commands in the example directory, to finalize the resolution of the example-specific dependencies:

```
npm install
```

You are now ready to start the sample code with Node.js. The easiest way to do it is to use the following command, replacing the [...] by the arguments that you want to pass to the demo code:

```
npm run demo [...]
```

This command, defined in `package.json`, will first start the TypeScript compiler using the simple `tsc` command, then run the transpiled code in Node.js.

The compilation uses the parameters specified in the file `tsconfig.json`, and produces

- a JavaScript file named `demo.js`, that Node.js can run
- a debug file named `demo.js.map`, that will help Node.js to locate the source of errors in the original TypeScript source file rather than reporting them in the JavaScript compiled file.

Note that the `package.json` file in our examples uses a relative reference to the local copy of the library, to avoid duplicating the library in each example. But of course, for your application, you can refer to the package directly in npm repository, by adding it to your project using the command:

```
npm install yoctolib-cjs
```

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-RS232-C**. You will find there an HTML file named `app.html`, and a TypeScript file `app.ts` 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.

No installation is needed to run this example, as the TypeScript library is referenced using a relative path. However, in order to allow the browser to run the code, the HTML page must be served by a Web server. We therefore provide a simple test server for this purpose, that you can start with the command:

```
npm run app-server
```

This command will compile the TypeScript sample code, make it available via an HTTP server on port 3000 and open a Web browser on this example. If you change the example source code, the TypeScript compiler will automatically be triggered to update the transpiled code and a simple page reload on the browser will make it possible to test the change.

As for the Node.js example, the compilation process will create a source map file which makes it possible to debug the example code in TypeScript source form within the browser debugger. Note that as of the writing of this document, this works on Chromium-based browsers but not yet in Firefox.

15.4. Control of the module part

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

```
import { YAPI, YErrorMsg, YModule } from 'yoctolib-cjs/yocto_api_nodejs.js';

async function startDemo(args: string[]): Promise<void>
{
    await YAPI.LogUnhandledPromiseRejections();

    // Setup the API to use the VirtualHub on local machine
    let errmsg: YErrorMsg = 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 device to use
    let module: YModule = 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:      '+
            ((await module.get_upTime())/1000)>>0) +' 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 < 3) {
    console.log("usage: npm run demo <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 methods, 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()` method. 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 { YAPI, YErrorMsg, YModule } from 'yoctolib-cjs/yocto_api_nodejs.js';

async function startDemo(args: string[]): Promise<void>
{
    await YAPI.LogUnhandledPromiseRejections();

    // Setup the API to use the VirtualHub on local machine
    let errmsg: YErrorMsg = 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 device to use
    let module: YModule = YModule.FindModule(args[0]);
    if(await module.isOnline()){
        if(args.length > 1) {
            let newname: string = 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 < 3) {
    console.log("usage: npm run demo <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()` method only 100000 times in the life of the module. Make sure you do not call this method within a loop.

Listing the modules

Obtaining the list of the connected modules is performed with the `YModule.FirstModule()` method which returns the first module found. Then, you only need to call the `nextModule()` method 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 { YAPI, YErrorMsg, YModule } from 'yoctolib-cjs/yocto_api_nodejs.js';

async function startDemo(): Promise<void>
{
    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');
        return;
    }
    refresh();
}

async function refresh(): Promise<void>
{
    try {
        let errmsg: YErrorMsg = new YErrorMsg();
        await YAPI.UpdateDeviceList(errmsg);

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

startDemo();
```

15.5. Error handling

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

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

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

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

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

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

16. Using Yocto-RS232-C 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` anymore now that `async / await` are part of the standard.

16.1. 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.

16.2. Using Yoctopuce library for JavaScript / EcmaScript 2017

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

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

- The Javascript / EcmaScript 2017 programming library¹
- VirtualHub² for Windows, macOS or Linux, depending on your OS

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

Using the official Yoctopuce library for node.js

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

¹ www.yoctopuce.com/EN/libraries.php

² www.yoctopuce.com/EN/virtualhub.php

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

```
npm install
```

Once done, you can start the example file using:

```
node demo.js
```

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

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

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

Using the Yoctopuce library within a browser (HTML)

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

Using the Yoctopuce library on older JavaScript engines

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

```
npm install -g babel-cli
npm install 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);
```

16.3. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS232-C. Here is the skeleton of a JavaScript code snippet to use the SerialPort function.

```
// For Node.js, we use function require()
// For HTML, we would use <script src="...">
require('yoctolib-es2017/yocto_api.js');
require('yoctolib-es2017/yocto_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("RS232M1C-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-RS232-C. 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-RS232-C module with serial number `RS232M1C-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("RS232M1C-123456.serialPort")
serialport = YSerialPort.FindSerialPort("RS232M1C-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.

reset

The `reset()` method of the objet returned by `YSerialPort.FindSerialPort` clear all internal buffers of the serial port.

readLine

The `readLine()` method returns the next CR/LF terminated line received on the serial port, or an empty string if no more lines are found (this function is non-blocking).

writeLine

The `writeLine()` method puts the specified string in the serial port output buffer, followed by CR/LF.

A real example, for Node.js

Open a command window (a terminal, a shell...) and go into the directory **example_nodejs/Doc-GettingStarted-Yocto-RS232-C** 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-RS232-C is not connected on the host running the browser, replace in the example the address `127.0.0.1` with the IP address of the host on which the Yocto-RS232-C is connected and where you run the VirtualHub.

```
"use strict";

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

let serialPort;

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 anysensor = YSerialPort.FirstSerialPort();
    if (anysensor) {
      let module = await anysensor.module();
      serial = await module.get_serialNumber();
    } else {
      console.log('No matching module connected, check cable !');
      return;
    }
  }
  console.log('Using device ' + serial);
  serialPort = YSerialPort.FindSerialPort(serial + ".serialPort");

  await serialPort.set_serialMode("9600,8N1");
  await serialPort.set_protocol("Line");
  await serialPort.reset();

  const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
  });
  console.log('Type line to send, or Ctrl-C to exit:');
  rl.on('line', (input) => {
    serialPort.writeLine(input);
  });

  refresh();
}

async function refresh() {
  let line;
  if (await serialPort.isOnline()) {
    do {
      line = await serialPort.readLine();
      if (line !== "") {
        console.log("Received: " + line);
      }
    } while (line !== "");
  } else {
    console.log('Module not connected');
  }
  setTimeout(refresh, 500);
}

startDemo();
```

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

```
npm install
```

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

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

Same example, but this time running in a browser

If you want to see how to use the library within a browser rather than with Node.js, switch to the directory **example_html/Doc-GettingStarted-Yocto-RS232-C**. 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;

  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;
    }

    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.FindSerialPort(serial + ".serialPort");

    await serialPort.set_serialMode("9600,8N1");
    await serialPort.set_protocol("Line");
    await serialPort.reset();

    refresh();
  }

  async function refresh()
  {
    let line;
    if (await serialPort.isOnline()) {
      line = await serialPort.readLine();
      document.getElementById('returned_value').value +=line;
    } else {
      console.log('Module not connected');
    }
    setTimeout(refresh, 500);
  }

  window.send_data = function()
  {
    var tosend = document.getElementById('line_input').value;
```

```

        serialPort.writeLine(tosend);
    };

    startDemo();
</script>
</head>
<body>
Module to use: <input id='serial'><br>
Type line to send<br>
<input id="line_input" onchange='javascript:send_data()'><br>
Received: <br/>
<textarea id="returned_value"></textarea>
</body>
</html>

```

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

16.4. 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);
}

```

16.5. 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 `ClassName.STATE_INVALID` value,

a `get_currentValue` method returns a `ClassName.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 Yocto-RS232-C 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 VirtualHub on the machine on which your modules are connected.

To start your tests with PHP, you need a PHP 7.1 (or more recent) server¹, preferably locally on you machine. If you wish to use the PHP server of your internet provider, it is possible, but you will probably need to configure your ADSL router for it to accept and forward TCP request on the 4444 port.

17.1. Getting ready

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

- The PHP programming library²
- VirtualHub³ for Windows, macOS, or Linux, depending on your OS

Our PHP library is based on PHP 8.x. In other words, our library works perfectly with any version of PHP currently still supported. However, in order not to abandon our customers with older installations, we maintain a version compatible with PHP 7.1. which dates back to 2016.

We also offer a version of the library that follows PSR's recommendations. For simplicity's sake, this version uses the same code as the php8 version, but each class is stored in a separate file. In addition, this version uses a `Yoctopuce\YoctoAPI` namespace. These changes make our library much easier to use with autoload installations.

Note that the examples in the documentation do not use the PSR version.

¹ A couple of free PHP servers: easyPHP for Windows, MAMP for macOS.

² www.yoctopuce.com/EN/libraries.php

³ www.yoctopuce.com/EN/virtualhub.php

In the library archive, there are thus three subdirectories:

- php7
- php8
- phpPSR

Choose the right directory according to the version of the library you wish to use, unzip the files of this directory into a directory of your choice accessible to your web server, plug in your modules, launch VirtualHub, and you are ready to start testing. You do not need to install any driver.

17.2. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS232-C. Here is the skeleton of a PHP code snippet to use the SerialPort function.

```
include('yocto_api.php');
include('yocto_serialport.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("RS232M1C-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-RS232-C.

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-RS232-C module with serial number `RS232M1C-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("RS232M1C-123456.serialPort");
$serialport = YSerialPort::FindSerialPort("RS232M1C-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.

reset

The `reset()` method of the object returned by `yFindSerialPort` clear all internal buffers of the serial port.

readLine

The `readLine()` method returns the next CR/LF terminated line received on the serial port, or an empty string if no more lines are found (this function is non-blocking).

writeLine

The `writeLine()` method puts the specified string in the serial port output buffer, followed by CR/LF.

A real example

Open your preferred text editor⁴, copy the code sample below, save it with the Yoctopuce library files in a location which is accessible to you web server, then use your preferred web browser to access this page. The code is also provided in the directory **Examples/Doc-GettingStarted-Yocto-RS232-C** 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
    include('../..'/php8/yocto_api.php');
    include('../..'/php8/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 $address (check USB cable)");
    print('Type line to send<br>');
    print("<input name='tosend'>");
    if(isset($_GET["tosend"])) {
      $tosend = $_GET["tosend"];
      $serialPort->writeLine($tosend);
      YAPI::Sleep(500);
      do {
        $line = $serialPort->readLine();
        if($line != "") {
          print("Received: " . $line . "<br/>");
        }
      } while ($line != '');
    } else {
      $serialPort->set_serialMode("9600,8N1");
      $serialPort->set_protocol("Line");
      $serialPort->reset();
    }
  }
</FORM>
</BODY>
</HTML>
```

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

```

    }
    YAPI::FreeAPI();
?>
<input type='submit'>

</FORM>
</BODY>
</HTML>

```

17.3. 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('../php8/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>
</HEAD>
<BODY>
  <FORM method='get'>
    <?php
      include('../php8/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();
    <?php
  </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.

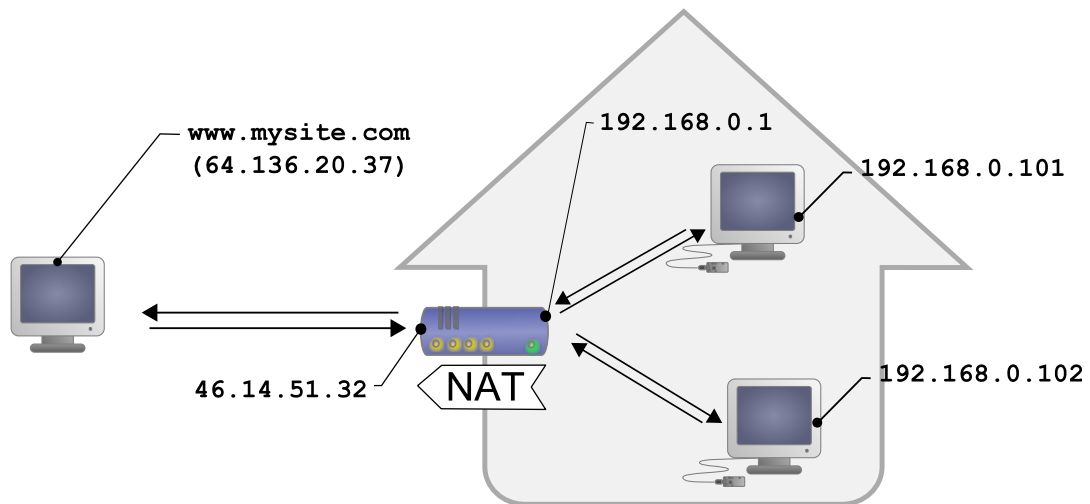
```
<HTML>
<HEAD>
  <TITLE>inventory</TITLE>
</HEAD>
<BODY>
<H1>Device list</H1>
<TT>
  <?php
    include('../..//php8/yocto_api.php');
    YAPI::RegisterHub("http://127.0.0.1:4444/");
    $module = YModule::FirstModule();
    while (!is_null($module)) {
      printf("%s (%s)<br>\n", $module->get_serialNumber(),
        $module->get_productName());
      $module=$module->nextModule();
    }
    YAPI::FreeAPI();
  ?>
</TT>
</BODY>
</HTML>
```

17.4. HTTP callback API and NAT filters

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

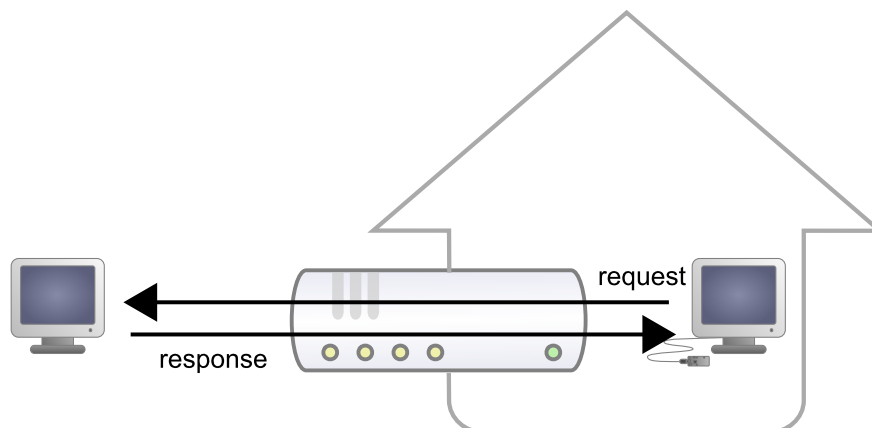
The NAT filter: advantages and disadvantages

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

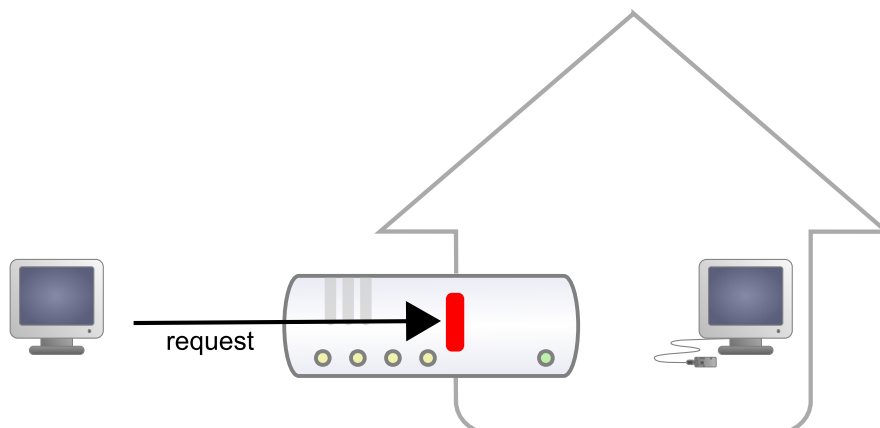


Typical DSL configuration: LAN machines are isolated from the outside by the DSL router

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



Responses from request from LAN machines are routed.

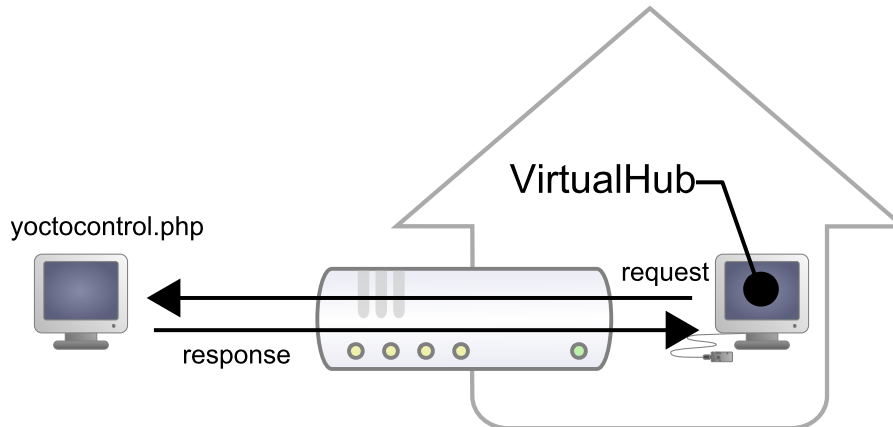


But requests from the outside are blocked.

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

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

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



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

Configuration

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

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

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

Serial	Logical Name	Description	Action
VIRTHUB0-7d1a86fb0		VirtualHub	configure view log file
RELAYHI1-00055		Yocto-PowerRelay	configure view log file beacon
TMPSENS1-05E7F		Yocto-Temperature	configure view log file beacon

Click on the "configure" button on the first line

VIRTHUB0-7d1a86fb09

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

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

Incoming connections

Authentication to read information from the devices: NO [edit](#)
Authentication to make changes to the devices: NO [edit](#)

Outgoing callbacks

Callback URL: octoHub [edit](#)
Delay between callbacks: min: 3 [s] max: 600 [s]

[Save](#) [Cancel](#)

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

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

1. Specify the Type of callback you want to use: **Yocto-API callback**

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

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

Callback URL:

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

Username:
 Password:

4. Setup the desired frequency of notifications:

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

5. Press on the **Test** button to check your parameters.

6. When everything works, press on the **OK** button.

And select "Yocto-API callback".

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

Usage

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

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

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

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

Common issues

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

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

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

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

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

Limitations

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

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

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 `ClassName.STATE_INVALID` value, a `get_currentValue` method returns a `ClassName.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 Yocto-RS232-C 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#. We support Visual Studio 2017 and its more recent versions.

18.1. Installation

Download the Visual Basic Yoctopuce library from the Yoctopuce web site¹. 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.

18.2. 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 modules². The source files in Visual Basic manage the high level part of the API. Therefore, you need both this DLL and the .vb files of the `sources` directory to create a project managing Yoctopuce modules.

Configuring a Visual Basic project

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

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

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

¹ www.yoctopuce.com/EN/libraries.php

² 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` directory³. 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.

18.3. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS232-C. Here is the skeleton of a Visual Basic code snippet to use the `SerialPort` function.

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

' Retrieve the object used to interact with the device
Dim serialport As YSerialPort
serialport = YSerialPort.FindSerialPort("RS232M1C-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-RS232-C module with serial number `RS232M1C-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("RS232M1C-123456.serialPort")
serialport = YSerialPort.FindSerialPort("RS232M1C-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.

³ Remember to change the filter of the selection window, otherwise the DLL will not show.

reset

The `reset()` method of the object returned by `yFindSerialPort` clear all internal buffers of the serial port.

readLine

The `readLine()` method returns the next CR/LF terminated line received on the serial port, or an empty string if no more lines are found (this function is non-blocking).

writeLine

The `writeLine()` method puts the specified string in the serial port output buffer, followed by CR/LF.

A real example

Launch Microsoft VisualBasic and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-RS232-C** of the Yoctopuce library.

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

```
Module Module1

    Private Sub Usage()
        Dim ex = System.AppDomain.CurrentDomain.FriendlyName
        Console.WriteLine("Usage")
        Console.WriteLine(ex + " <serial_number> <frequency> <dutyCycle>")
        Console.WriteLine(ex + " <logical_name> <frequency> <dutyCycle>")
        Console.WriteLine(ex + " any <frequency> <dutyCycle> (use any discovered device)")
        Console.WriteLine("    <frequency>: integer between 1Hz and 1000000Hz")
        Console.WriteLine("    <dutyCycle>: floating point number between 0.0 and 100.0")
        System.Threading.Thread.Sleep(2500)
    End Sub
End Sub

Sub Main()
    Dim argv() As String = System.Environment.GetCommandLineArgs()
    Dim errmsg As String = ""
    Dim target As String
    Dim serialPort As YSerialPort
    Dim line As String

    If (YAPI.RegisterHub("usb", errmsg) <> YAPI.SUCCESS) Then
        Console.WriteLine("RegisterHub error: " + errmsg)
        Environment.Exit(0)
    End If
    If (argv.Length > 1) Then
        target = argv(1)
        serialPort = YSerialPort.FindSerialPort(target + ".serialPort")
        If (Not serialPort.IsOnline()) Then
            Console.WriteLine("No module connected (check cable)")
            Environment.Exit(0)
        End If
    Else
        serialPort = YSerialPort.FirstSerialPort()
        If (serialPort Is Nothing) Then
            Console.WriteLine("No module connected (check USB cable)")
            Environment.Exit(0)
        End If
    End If
    serialPort.set_serialMode("9600,8N1")
    serialPort.set_protocol("Line")
    serialPort.reset()
    Do
        YAPI.Sleep(500, errmsg)
        Do
            line = serialPort.readLine()
            If (line <> "") Then
                Console.WriteLine("Received: " + line)
            End If
        Loop While (line <> "")
        Console.WriteLine("Type line to send, or Ctrl-C to exit: ")
    End Do
End Sub
```

```

        line = Console.ReadLine()
        serialPort.WriteLine(line)
    Loop While (line <> "")
    YAPI.FreeAPI()
End Sub

End Module

```

18.4. Control of the module part

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

```

Imports System.IO
Imports System.Environment

Module Module1

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

    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 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.WriteLine("beacon:      ")
            If (m.get_beacon() = Y_BEACON_ON) Then
                Console.WriteLine("ON")
            Else
                Console.WriteLine("OFF")
            End If
            Console.WriteLine("upTime:      " + Str(m.get_upTime() / 1000) + " sec")
            Console.WriteLine("USB current:  " + Str(m.get_usbCurrent()) + " mA")
            Console.WriteLine("Logs:")
            Console.WriteLine(m.get_lastLogs())
        Else
            Console.WriteLine(argv(1) + " not connected (check identification and USB cable)")
        End If
        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 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 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 If
            m.set_logicalName(newname)
            m.saveToFlash() REM do not forget this
            Console.WriteLine("Module: serial= " + m.get_serialNumber())
            Console.WriteLine(" / name= " + m.get_logicalName())
        Else
            Console.WriteLine("not connected (check identification and USB cable)")
        End If
        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

```

18.5. 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 `ClassName.STATE_INVALID` value, a `get_currentValue` method returns a `ClassName.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.

19. Using Yocto-RS232-C with Delphi or Lazarus

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

Lazarus² is a free IDE based on Free-Pascal, it has nothing to envy to Delphi and is available for both Windows and Linux. The Yoctopuce Delphi library is compatible with both Windows and Linux versions of Lazarus

Delphi libraries are provided not as VCL components, but directly as source files. These files are compatible with most Delphi and Lazarus versions.³

19.1. Preparation

Go to the Yoctopuce web site and download the Yoctopuce Delphi libraries⁴. Uncompress everything in a directory of your choice.

- With Delphi, add the subdirectory *sources* in the list of directories of Delphi libraries.⁵
- With Lazarus, open your project options and add the *sources* folder to your "other unit files" path.⁶

Windows

With Windows, the Yoctopuce Delphi / Lazarus library uses two dlls *yapi.dll* (32-bit version) and *yapi64.dll* (64-bit version). All the applications that you create with Delphi or Lazarus must have access to these DLL. The simplest way to ensure this is to make sure that they are located in the same directory as the executable file of your application. You can find these dlls in the *sources/dll* folder.

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 www.lazarus-ide.org

³ Delphi libraries are regularly tested with Delphi 5, Delphi XE2, and the latest version of Lazarus.

⁴ www.yoctopuce.com/EN/libraries.php

⁵ Use the **Tools / Environment options** menu.

⁶ Use the Menu **Project / Project options/ Compiler options / Paths**

Linux

Under Linux, the Delphi / Lazarus library uses the following lib files:

- *libyapi-i386.so* for Intel 32-bit systems
- *libyapi-amd64.so* for Intel 64-bit systems
- *libyapi-armhf.so* for ARM 32-bit systems
- *libyapi-aarch64.so* for ARM 64-bit systems

You will find these lib files in the *sources/dll* folder. You have to make sure that

- Lazarus can find the right .so file at compilation time.
- The executable can find it at execution time.

The simplest way to ensure this is to copy all four .so files into the */usr/lib* folder. Alternatively, you can copy them next to your main source file and adjust your *LD_LIBRARY_PATH* environment variable accordingly.

19.2. About examples

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.

Note that most of these examples use command line parameters ⁷.

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.

19.3. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS232-C. Here is the skeleton of a Delphi code snippet to use the SerialPort function.

```
uses yocto_api, yocto_serialport;

var errmsg: string;
    serialport: TYSerialPort;

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

// Retrieve the object used to interact with the device
serialport = yFindSerialPort("RS232M1C-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-RS232-C.

⁷ See <https://www.yoctopuce.com/EN/article/about-programming-examples>

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-RS232-C module with serial number *RS232M1C-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("RS232M1C-123456.serialPort");
serialport := yFindSerialPort("RS232M1C-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.

reset

The `reset()` method of the object returned by `yFindSerialPort` clear all internal buffers of the serial port.

readLine

The `readLine()` method returns the next CR/LF terminated line received on the serial port, or an empty string if no more lines are found (this function is non-blocking).

writeLine

The `writeLine()` method puts the specified string in the serial port output buffer, followed by CR/LF.

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;
{$APPTYPE CONSOLE}
uses
  SysUtils,
  {$IFDEF UNIX}
  windows,
  {$ENDIF UNIX}
  yocto_api,
  yocto_serialport;

var
  errmsg, target, channel: string;
  serialport: TYserialport;
  line: string;

begin
```

```

if (YRegisterHub('usb', errmsg) <> YAPI_SUCCESS) then
begin
  writeln('RegisterHub error: ' + errmsg);
  halt;
end;

if (paramcount>1) then
begin
  target := paramstr(1);
  serialport := YFindSerialPort(target + '.serialPort');
end
else
begin
  serialport := YFirstSerialPort();
  if (serialport = nil) then
  begin
    writeln('No module connected (check USB cable)');
    sleep(3000);
    halt;
  end;
end;
serialport.set_serialMode('9600,8N1');
serialport.set_protocol('Line');
serialport.reset();
repeat
  ySleep(500, errmsg);
  repeat
    line := serialport.readLine();
    if (line <> '') then
    begin
      writeln('Received: ' + line);
    end;
  until (line = '');
  writeln('Type line to send, or Ctrl-C to exit: ');
  readln(line);
  serialport.writeLine(line);
until (line = '');
yFreeAPI();
end.

```

19.4. 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.

```

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

const
  serial = 'RS232M1C-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

```

```

end;

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

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

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

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

    repeat
        read(c);
        case c of
            'r': refresh(module);
            'b': beacon(module,Y_BEACON_ON);
            ' ': beacon(module,Y_BEACON_OFF);
        end;
    until c = 'x';
    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 = 'RS232M1C-123456'; // use serial number or logical name

var
    module : TModule;
    errmsg : string;
    newname : string;

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

    module := yFindModule(serial);
    if (not(module.isOnline)) then
    begin

```

```

        writeln('Module not connected (check identification and USB cable)');
        exit;
    end;

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

```

19.5. 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 `ClassName.STATE_INVALID` value, a `get_currentValue` method returns a `ClassName.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.

20. Using the Yocto-RS232-C with Universal Windows Platform

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

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

20.1. Blocking and asynchronous functions

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

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

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

Example:

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

    return result;
}

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

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

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

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

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

20.2. Installation

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

20.3. 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-RS232-C, here is what you must add in the "Capabilities" node:

```
<DeviceCapability Name="humaninterfacedevice">
  <!-- Yocto-RS232-C -->
  <Device Id="vidpid:24e0 00CE">
    <Function Type="usage:ff00 0001" />
  </Device>
</DeviceCapability>
```

² www.yoctopuce.com/EN/libraries.php

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

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

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

20.4. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS232-C. 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("RS232M1C-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-RS232-C module with serial number *RS232M1C-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("RS232M1C-123456.serialPort");
serialport = YSerialPort.FindSerialPort("RS232M1C-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.

reset

The `reset()` method of the objet returned by `YSerialPort.FindSerialPort` clear all internal buffers of the serial port.

readLine

The `readLine()` method returns the next CR/LF terminated line received on the serial port, or an empty string if no more lines are found (this function is non-blocking).

writeLine

The `writeLine()` method puts the specified string in the serial port output buffer, followed by CR/LF.

20.5. A real example

Launch Visual Studio and open the corresponding project provided in the directory **Examples/Doc-GettingStarted-Yocto-RS232-C** 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 ToSend { get; set; }

        public override async Task<int> Run()
        {
            try {
                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");
                }

                await serialPort.set_serialMode("9600,8N1");
                await serialPort.set_protocol("Line");
                await serialPort.reset();
                string line;
                do {
                    if (ToSend != "") {
                        await serialPort.writeLine(ToSend);
                        ToSend = "";
                    }
                    await YAPI.Sleep(500);
                } do {
                    line = await serialPort.readLine();
                    if (line != "") {
                        WriteLine("Received: " + line);
                    }
                }
            }
        }
    }
}
```

```

        } while (line != "");

        } while (line != "");
    } catch (YAPI_Exception ex) {
        WriteLine("error: " + ex.Message);
    }

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

```

20.6. 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)");
            }
            await YAPI.FreeAPI();
            return 0;
        }
    }
}

```

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

Changing the module settings

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

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

```

20.7. 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);
}

```


21. Using Yocto-RS232-C with Objective-C

Objective-C is language of choice for programming on macOS, due to its integration with the Cocoa framework. Yoctopuce supports the XCode versions supported by Apple. The Yoctopuce library is ARC compatible. You can therefore implement your projects either using the traditional *retain / release* method, or using the *Automatic Reference Counting*.

Yoctopuce Objective-C libraries¹ are integrally provided as source files. A section of the low-level library is written in pure C, but you should not need to interact directly with it: everything was done to ensure the simplest possible interaction from Objective-C.

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

In order to keep them simple, all the examples provided in this documentation are console applications. Naturally, the libraries function in a strictly identical manner if you integrate them in an application with a graphical interface. You can find on Yoctopuce blog a detailed example² with video shots showing how to integrate the library into your projects.

21.1. Control of the SerialPort function

A few lines of code are enough to use a Yocto-RS232-C. Here is the skeleton of a Objective-C code snippet 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:@"RS232M1C-123456.serialPort"];

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

¹ www.yoctopuce.com/EN/libraries.php

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

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-RS232-C.

[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-RS232-C module with serial number *RS232M1C-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:@"RS232M1C-123456.serialPort"];
YSerialPort *serialport = [SerialPort FindSerialPort:@"RS232M1C-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.

reset

The `reset()` method of the object returned by `YSerialPort.FindSerialPort` clear all internal buffers of the serial port.

readLine

The `readLine()` method returns the next CR/LF terminated line received on the serial port, or an empty string if no more lines are found (this function is non-blocking).

writeLine

The `writeLine()` method puts the specified string in the serial port output buffer, followed by CR/LF.

A real example

Launch Xcode 4.2 and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-RS232-C** 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[])
```

```

{
    @autoreleasepool {
        NSError *error;
        // 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;
            }
        }

        [serialPort set_serialMode:@"9600,8N1"];
        [serialPort set_protocol:@"Line"];
        [serialPort reset];

        NSString *line;
        do {
            char input[256] = {0};
            [YAPI Sleep:500 :&error];
            do {
                line = [serialPort readLine];
                if (![line isEqualToString:@""]) {
                    NSLog(@"Received: %@", line);
                }
            } while (![line isEqualToString:@""]);

            NSLog(@"Type line to send, or Ctrl-C to exit: ");
            fgets(input, sizeof(input), stdin);
            line = [NSString stringWithUTF8String:input];
            [serialPort writeLine:line];
        } while (![line isEqualToString:@"\n"]);

        [YAPI FreeAPI];
    }
    return 0;
}

```

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

Listing the modules

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

```

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

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

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

```

21.3. 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 `ClassName.STATE_INVALID` value, a `get_currentValue` method returns a `ClassName.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.

22. 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, and so on.

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

22.1. 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.

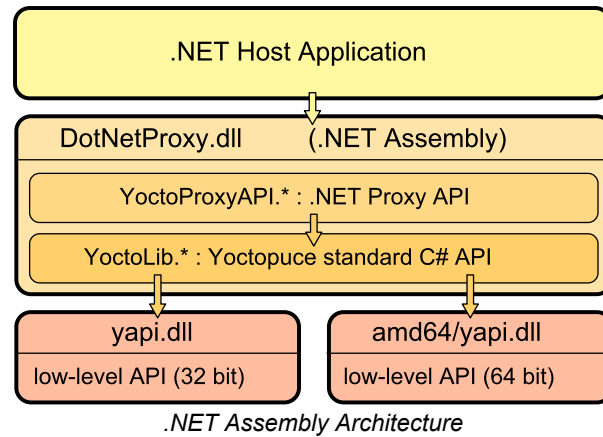
However, if the command line API is the easiest solution, it is neither the fastest nor the most efficient. For each call, the executable must initialize its own API and make an inventory of USB connected modules. This requires about one second per call.

22.2. .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 would not 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 cannot 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.



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('%0.3f %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 do not 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 the `YoctoLib` version directly, but not both!

Compatibility

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.

22.3. VirtualHub and HTTP GET

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, 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 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-RS232-C with the *RS232M1C-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/RS232M1C-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/RS232M1C-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/RS232M1C-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/RS232M1C-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/RS232M1C-12345/api/serialPort/logicalName
```

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

```
http://127.0.0.1:4444/bySerial/RS232M1C-12345/api/serialPort?logicalName=myFunction
```

You can find the list of available attributes for your Yocto-RS232-C 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/RS232M1C-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/RS232M1C-12345/dataLogger.json?id=serialPort&utc=1389801080
```

22.4. 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 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 result returned in the buffer variable respects the HTTP protocol. It therefore includes an HTTP header. This header ends with two empty lines, that is a sequence of four ASCII characters 13, 10, 13, 10.

Here is a sample program written in pascal using the *yapi.dll* DLL to read and then update the luminosity of a module.

```
// Dll functions import
function yapiInitAPI(mode:integer;
```

```

errmsg : pansichar):integer;cdecl;
external 'yapi.dll' name 'yapiInitAPI';
function yapiUpdateDeviceList(force:integer;errmsg : pansichar):integer;cdecl;
external 'yapi.dll' name 'yapiUpdateDeviceList';
function yapiHTTPRequest(device:pansichar,url:pansichar; buffer:pansichar;
buffsize:integer;var fullsize:integer;
errmsg : pansichar):integer;cdecl;
external 'yapi.dll' name 'yapiHTTPRequest';

var
errmsgBuffer : array [0..256] of ansichar;
dataBuffer : array [0..1024] of ansichar;
errmsg,data : pansichar;
fullsize,p : integer;

const
serial = 'RS232M1C-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_`.

22.5. 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.

23. Advanced programming

The preceding chapters have introduced, in each available language, the basic programming functions which can be used with your Yocto-RS232-C 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.

23.1. Event programming

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

Detecting module arrival and departure

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

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

The callback

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

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

Initialization

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

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

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

Triggering callbacks

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

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

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

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

Detecting a modification in the value of a sensor

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

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

Callback invocation

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

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

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

The value change callback

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

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

In most programming languages, callbacks are global procedures, not methods. If you wish for the callback to call a method of an object, define your callback as a global procedure which then calls your method. If you need to keep a reference to your object, you can store it directly in the `YGenericSensor` object using function `set_userData`. You can then retrieve it in the global callback procedure using `get_userData`.

Setting up a value change callback

The callback is set up for a given `GenericSensor` function with the help of the `registerValueCallback` method. The following example sets up a callback for the first available `GenericSensor` function.

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

Note that each module function can thus have its own distinct callback. By the way, if you like to work with value change callbacks, you will appreciate the fact that value change callbacks are not limited to sensors, but are also available for all Yoctopuce devices (for instance, you can also receive a callback any time a relay state changes).

The timed report callback

This type of callback is automatically called at a predefined time interval. The callback frequency can be configured individually for each sensor, with frequencies going from hundred calls per seconds down to one call per hour. The callback takes as parameter the relevant function and the measured value, as an `YMeasure` object. Contrarily to the value change callback that only receives the latest value, an `YMeasure` object provides both minimal, maximal and average values since the timed report callback. Moreover, the measure includes precise timestamps, which makes it possible to use timed reports for a time-based graph even when not handled immediately.

```
static void periodicCallback(YGenericSensor fct, YMeasure measure)
{
    Console.WriteLine(fct.get_hardwareId() + "=" +
        measure.get_averageValue());
}
```

Setting up a timed report callback

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

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

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

```
YGenericSensor f = YGenericSensor.FirstGenericSensor();
f.set_reportFrequency("4/m");
f.registerTimedReportCallback(periodicCallback);
```

As for value change callbacks, each module function can thus have its own distinct timed report callback.

Generic callback functions

It is sometimes desirable to use the same callback function for various types of sensors (e.g. for a generic sensor graphing application). This is possible by defining the callback for an object of class `YSensor` rather than `YGenericSensor`. Thus, the same callback function will be usable with any subclass of `YSensor` (and in particular with `YGenericSensor`). With the callback function, you can use the method `get_unit()` to get the physical unit of the sensor, if you need to display it.

A complete example

You can find a complete example implemented in your favorite programming language in the *Examples/Prog-EventBased* directory of the corresponding library.

23.2. The data logger

Your Yocto-RS232-C 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 it 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-RS232-C 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-RS232-C 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-RS232-C flash memory, you must call the `get_recordedData()` method of the desired sensor, and specify the time interval for which you want to retrieve measures. The time interval is given by the start and stop UNIX timestamp. You can also specify 0 if you don't want any start or stop limit.

The `get_recordedData()` method does not return directly an array of measured values, since in some cases it would cause a huge load that could affect the responsiveness of the application. Instead, this function will return an `YDataSet` object that can be used to retrieve immediately an overview of the measured data (summary), and then to load progressively the details when desired.

Here are the main methods used to retrieve recorded measures:

1. **dataset = sensor.get_recordedData(0,0):** select the desired time interval
2. **dataset.loadMore():** load data from the device, progressively
3. **dataset.get_summary():** get a single measure summarizing the full time interval
4. **dataset.get_preview():** get an array of measures representing a condensed version of the whole set of measures on the selected time interval (reduced by a factor of approx. 200)
5. **dataset.get_measures():** get an array with all detailed measures (that grows while `loadMore` is being called repeatedly)

Measures are instances of `YMeasure`². They store simultaneously the minimal, average and maximal value at a given time, that you can retrieve using methods **get_minValue()**, **get_averageValue()** and **get_maxValue()** respectively. Here is a small example that uses the functions above:

```
// We will retrieve all measures, without time limit
YDataSet dataset = sensor.get_recordedData(0, 0);

// First call to loadMore() loads the summary/preview
dataset.loadMore();
YMeasure summary = dataset.get_summary();
```

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-RS232-C does not have a battery, it cannot guess alone the current time when powered on. Nevertheless, the Yocto-RS232-C 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-RS232-C 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-RS232-C 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-RS232-C 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-RS232-C 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-RS232-C 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-RS232-C 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-RS232-C might be subject to a clock skew (theoretically up to 2%).

23.3. Sensor calibration

Your Yocto-RS232-C 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-RS232-C. 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-RS232-C 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-RS232-C. On the other hand, the API supports an external arbitrary calibration, implemented with callbacks.

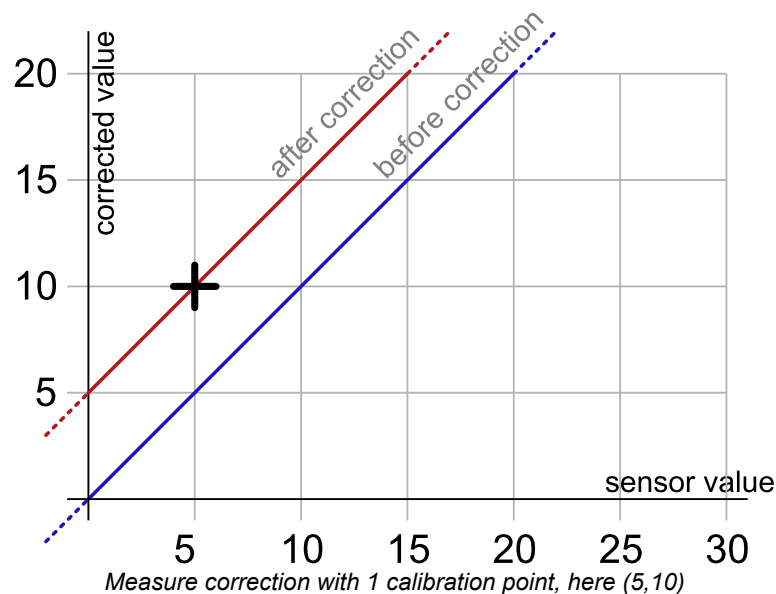
1 to 5 point linear interpolation

These transformations are performed directly inside the Yocto-RS232-C which means that you only have to store the calibration points in the module flash memory, and all the correction computations are done in a perfectly transparent manner: The function `get_currentValue()` returns the corrected value while the function `get_currentRawValue()` keeps returning the value before the correction.

Calibration points are simply (*Raw_value*, *Corrected_value*) couples. Let us look at the impact of the number of calibration points on the corrections.

1 point correction

The 1 point correction only adds a shift to the measures. For example, if you provide the calibration point (*a*, *b*), all the measured values are corrected by adding to them $b-a$, so that when the value read on the sensor is *a*, the `genericSensor1` function returns *b*.



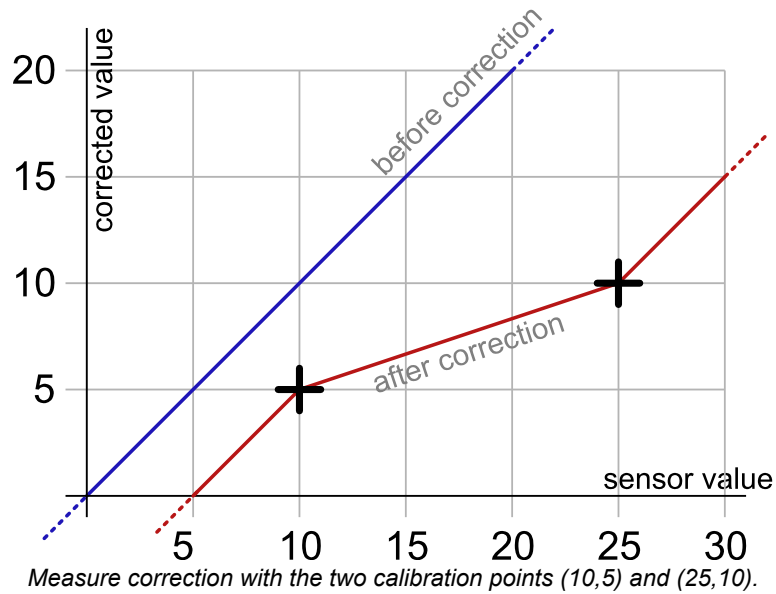
The application is very simple: you only need to call the `calibrateFromPoints()` method of the function you wish to correct. The following code applies the correction illustrated on the graph above to the first `genericSensor1` function found. Note the call to the `saveToFlash` method of the module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

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

2 point correction

2 point correction allows you to perform both a shift and a multiplication by a given factor between two points. If you provide the two points (*a*, *b*) and (*c*, *d*), the function result is multiplied $(d-b)/(c-a)$ in the [*a*, *c*] range and shifted, so that when the value read by the sensor is *a* or *c*, the `genericSensor1` function returns respectively *b* and *d*. Outside of the [*a*, *c*] range, the values are simply shifted, so as

to preserve the continuity of the measures: an increase of 1 on the value read by the sensor induces an increase of 1 on the returned value.



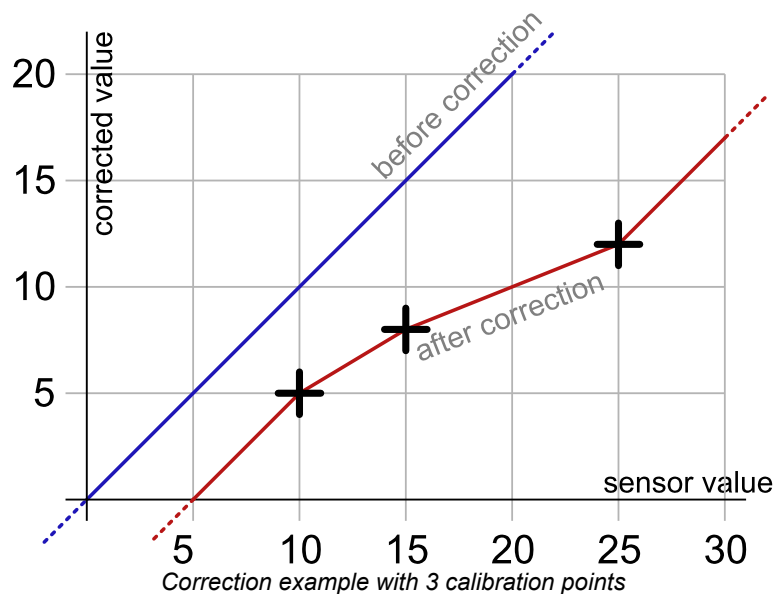
The code allowing you to program this calibration is very similar to the preceding code example.

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

Note that the values before correction must be sorted in a strictly ascending order, otherwise they are simply ignored.

3 to 5 point correction

3 to 5 point corrections are only a generalization of the 2 point method, allowing you to create up to 4 correction ranges for an increased precision. These ranges cannot be disjoint.



Back to normal

To cancel the effect of a calibration on a function, call the `calibrateFromPoints()` method with two empty arrays.

```
Double[] ValuesBefore = {};
Double[] ValuesAfter = {};
YGenericSensor f = YGenericSensor.FirstGenericSensor();
f.calibrateFromPoints(ValuesBefore, ValuesAfter);
f.get_module().saveToFlash();
```

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

Limitations

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

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

Arbitrary interpolation

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

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

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

24. Firmware Update

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

24.1. VirtualHub or the YoctoHub

It is possible to update the firmware directly from the web interface of VirtualHub or of a 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.

24.2. 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:\>
```

24.3. 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.

24.4. 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.

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");
}
...
```

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

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

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

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

Updating the firmware

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

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

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

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

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

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

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

```
YFirmwareUpdate fw_update = m.updateFirmware(newfirm);
....
int status = fw_update.startUpdate();
while (status < 100 && status >= 0) {
    int newstatus = fw_update.get_progress();
    if (newstatus != status) {
        Console.WriteLine(status + "% "
            + fw_update.get_progressMessage());
    }
}
```

```
YAPI.Sleep(500, ref errmsg);
status = newstatus;
}

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

An Android characteristic

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

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

24.5. The "update" mode

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

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

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

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

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

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

25. High-level API Reference

This chapter summarizes the high-level API functions to drive your Yocto-RS232-C. 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.

25.1. 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	<code>import com.yoctopuce.YoctoAPI.YAPI;</code>
dnf	<code>import YoctoProxyAPI.YAPIProxy</code>
cp	<code>#include "yocto_api_proxy.h"</code>
m1	<code>import YoctoProxyAPI.YAPIProxy"</code>
cpp	<code>#include "yocto_api.h"</code>
vb	<code>yocto_api.vb</code>
cs	<code>yocto_api.cs</code>
py	<code>from yocto_api import *</code>
php	<code>require_once('yocto_api.php');</code>
ts	<code>in HTML: import { YAPI, YErrorMsg, YModule, YSensor } from '../dist/esm/yocto_api_browser.js';</code> <code>in Node.js: import { YAPI, YErrorMsg, YModule, YSensor } from 'yoctolib-cjs/yocto_api_nodejs.js';</code>
ppy	<code>from yoctolib.yocto_api import *</code>
vi	<code>YModule.vi</code>

Global functions

YAPI.AddTrustedCertificates(*certificate*)

Adds a TLS/SSL certificate to the list of trusted certificates.

cpp vb cs java py php ts ppy

YAPI.AddUdevRule(*force*)

Adds a UDEV rule which authorizes all users to access Yoctopuce modules connected to the USB ports.

cpp vb cs java py php ts

YAPI.CheckLogicalName(*name*)

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

cpp vb cs java py php ts ppy

YAPI.ClearHTTPCallbackCacheDir(*removeFiles*)

Disables the HTTP callback cache.

java php

YAPI.DisableExceptions()

Disables the use of exceptions to report runtime errors.

cpp vb cs py php ts

YAPI.DownloadHostCertificate(*url*, *mstimeout*)

Download the TLS/SSL certificate from the hub.

cpp vb cs java py php ts ppy

YAPI.EnableExceptions()

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

cpp vb cs py php ts

YAPI.EnableUSBHost(osContext)

This function is used only on Android.

java

YAPI.FreeAPI()

Waits for all pending communications with Yoctopuce devices to be completed then frees dynamically allocated resources used by the Yoctopuce library.

cpp vb cs java py php ts dnp ppy

YAPI.GetAPIVersion()

Returns the version identifier for the Yoctopuce library in use.

cpp vb cs java py php ts dnp ppy

YAPI.GetCacheValidity()

Returns the validity period of the data loaded by the library.

cpp vb cs java py php ts ppy

YAPI.GetDeviceListValidity()

Returns the delay between each forced enumeration of the used YoctoHubs.

cpp vb cs java py php ts ppy

YAPI.GetDllArchitecture()

Returns the system architecture for the Yoctopuce communication library in use.

dnp

YAPI.GetDllPath()

Returns the paths of the DLLs for the Yoctopuce library in use.

dnp

YAPI.GetLog(lastLogLine)

Retrieves Yoctopuce low-level library diagnostic logs.

dnp

YAPI.GetNetworkTimeout()

Returns the network connection delay for `yRegisterHub()` and `yUpdateDeviceList()`.

cpp vb cs java py php ts dnp ppy

YAPI.GetTickCount()

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

cpp vb cs java py php ts ppy

YAPI.GetYAPISharedLibraryPath()

Returns the path to the dynamic YAPI library.

cpp vb cs java py php ts ppy

YAPI.HandleEvents(errmsg)

Maintains the device-to-library communication channel.

cpp vb cs java py php ts ppy

YAPI.InitAPI(mode, errmsg)

Initializes the Yoctopuce programming library explicitly.

cpp vb cs java py php ts ppy

YAPI.PreregisterHub(url, errmsg)

Fault-tolerant alternative to `yRegisterHub()`.

`cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `ppy`

YAPI.RegisterDeviceArrivalCallback(arrivalCallback)

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

`cpp` `vb` `cs` `java` `py` `php` `ts` `ppy`

YAPI.RegisterDeviceRemovalCallback(removalCallback)

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

`cpp` `vb` `cs` `java` `py` `php` `ts` `ppy`

YAPI.RegisterHub(url, errmsg)

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

`cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `ppy`

YAPI.RegisterHubDiscoveryCallback(hubDiscoveryCallback)

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

`cpp` `vb` `cs` `java` `py` `ts` `ppy`

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.

`cpp` `vb` `cs` `java` `py` `ts` `ppy`

YAPI.SelectArchitecture(arch)

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

`py`

YAPI.SetCacheValidity(cacheValidityMs)

Change the validity period of the data loaded by the library.

`cpp` `vb` `cs` `java` `py` `php` `ts` `ppy`

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.

`cpp` `vb` `cs` `java` `py` `php` `ts` `ppy`

YAPI.SetHTTPCallbackCacheDir(directory)

Enables the HTTP callback cache.

`java` `php`

YAPI.SetNetworkSecurityOptions(opts)

Enables or disables certain TLS/SSL certificate checks.

`cpp` `vb` `cs` `java` `py` `php` `ts` `ppy`

YAPI.SetNetworkTimeout(networkMsTimeout)

Modifies the network connection delay for `yRegisterHub()` and `yUpdateDeviceList()`.

`cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `ppy`

YAPI.SetTimeout(callback, ms_timeout, args)

Invoke the specified callback function after a given timeout.

ts

YAPI.SetTrustedCertificatesList(certificatePath)

Set the path of Certificate Authority file on local filesystem.

cpp vb cs java py php ts ppy

YAPI.SetUSBPacketAckMs(pktAckDelay)

Enables the acknowledge of every USB packet received by the Yoctopuce library.

java

YAPI.Sleep(ms_duration, errmsg)

Pauses the execution flow for a specified duration.

cpp vb cs java py php ts ppy

YAPI.TestHub(url, mtimeout, errmsg)

Test if the hub is reachable.

cpp vb cs java py php ts dnp ppy

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.

cpp vb cs java py ts ppy

YAPI.UnregisterHub(url)

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

cpp vb cs java py php ts ppy

YAPI.UpdateDeviceList(errmsg)

Triggers a (re)detection of connected Yoctopuce modules.

cpp vb cs java py php ts ppy

YAPI.UpdateDeviceList_async(callback, context)

Triggers a (re)detection of connected Yoctopuce modules.

25.2. 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:

cpp	<code>#include "yocto_api.h"</code>
vb	<code>yocto_api.vb</code>
cs	<code>yocto_api.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YModule;</code>
py	<code>from yocto_api import *</code>
php	<code>require_once('yocto_api.php');</code>
ts	<code>in HTML: import { YAPI, YErrorMsg, YModule, YSensor } from '../dist/esm/yocto_api_browser.js'; in Node.js: import { YAPI, YErrorMsg, YModule, YSensor } from 'yoctolib-cjs/yocto_api_nodejs.js';</code>
dnf	<code>import YoctoProxyAPI.YModuleProxy</code>
cp	<code>#include "yocto_module_proxy.h"</code>
ppy	<code>from yoctolib.yocto_api import *</code>
vi	<code>YModule.vi</code>
ml	<code>import YoctoProxyAPI.YModuleProxy"</code>

Global functions

YModule.FindModule(func)

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

cpp vb cs java py php ts dnf ppy

YModule.FindModuleInContext(yctx, func)

Retrieves a module for a given identifier in a YAPI context.

java ts ppy

YModule.FirstModule()

Starts the enumeration of modules currently accessible.

cpp vb cs java py php ts ppy

YModule properties

module→Beacon [writable]

State of the localization beacon.

dnf

module→FirmwareRelease [read-only]

Version of the firmware embedded in the module.

dnf

module→FunctionId [read-only]

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

dnf

module→HardwareId [read-only]

Unique hardware identifier of the module.

dnf

module→IsOnline *[read-only]*

Checks if the module is currently reachable.

dnp

module→LogicalName *[writable]*

Logical name of the module.

dnp

module→Luminosity *[writable]*

Luminosity of the module informative LEDs (from 0 to 100).

dnp

module→ProductId *[read-only]*

USB device identifier of the module.

dnp

module→ProductName *[read-only]*

Commercial name of the module, as set by the factory.

dnp

module→ProductRelease *[read-only]*

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

dnp

module→SerialNumber *[read-only]*

Serial number of the module, as set by the factory.

dnp

YModule methods**module→addFileToHTTPCallback**(filename)

Adds a file to the uploaded data at the next HTTP callback.

cmd cpp vb cs java py php ts dnp

module→checkFirmware(path, onlynew)

Tests whether the bin file is valid for this module.

cmd cpp vb cs java py php ts dnp ppy

module→clearCache()

Invalidates the cache.

cpp vb cs java py php ts ppy

module→describe()

Returns a descriptive text that identifies the module.

cpp vb cs java py php ts

module→download(pathname)

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

cmd cpp vb cs java py php ts dnp ppy

module→functionBaseType(functionIndex)Retrieves the base type of the *n*th function on the module.

cpp vb cs java py php ts

module→functionCount()

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

cpp vb cs java py php ts

module→functionId(functionIndex)

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

cpp vb cs java py php ts

module→functionName(functionIndex)

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

cpp vb cs java py php ts

module→functionType(functionIndex)

Retrieves the type of the *n*th function on the module.

cpp vb cs java py php ts

module→functionValue(functionIndex)

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

cpp vb cs java py php ts

module→get_allSettings()

Returns all the settings and uploaded files of the module.

cmd cpp vb cs java py php ts dnp ppy

module→get_beacon()

Returns the state of the localization beacon.

cmd cpp vb cs java py php ts dnp ppy

module→get_errorMessage()

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

cpp vb cs java py php ts ppy

module→get_errorType()

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

cpp vb cs java py php ts ppy

module→get_firmwareRelease()

Returns the version of the firmware embedded in the module.

cmd cpp vb cs java py php ts dnp ppy

module→get_functionIds(funType)

Retrieve all hardware identifier that match the type passed in argument.

cmd cpp vb cs java py php ts dnp

module→get_hardwareId()

Returns the unique hardware identifier of the module.

cpp vb cs java py php ts dnp ppy cmd

module→get_icon2d()

Returns the icon of the module.

cmd cpp vb cs java py php ts dnp ppy

module→get_lastLogs()

Returns a string with last logs of the module.

cmd cpp vb cs java py php ts dnp ppy

module→get_logicalName()

Returns the logical name of the module.

cpp vb cs java py php ts dnp ppy cmd

module→get_luminosity()

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

cmd cpp vb cs java py php ts dnp ppy

module→get_parentHub()

Returns the serial number of the YoctoHub on which this module is connected.

cmd cpp vb cs java py php ts dnp ppy

module→get_persistentSettings()

Returns the current state of persistent module settings.

cmd cpp vb cs java py php ts dnp ppy

module→get_productId()

Returns the USB device identifier of the module.

cmd cpp vb cs java py php ts dnp ppy

module→get_productName()

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

cmd cpp vb cs java py php ts dnp ppy

module→get_productRelease()

Returns the release number of the module hardware, preprogrammed at the factory.

cmd cpp vb cs java py php ts dnp ppy

module→get_rebootCountdown()

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

cmd cpp vb cs java py php ts dnp ppy

module→get_serialNumber()

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

cpp vb cs java py php ts dnp ppy cmd

module→get_subDevices()

Returns a list of all the modules that are plugged into the current module.

cmd cpp vb cs java py php ts dnp

module→get_upTime()

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

cmd cpp vb cs java py php ts dnp ppy

module→get_url()

Returns the URL used to access the module.

cmd cpp vb cs java py php ts dnp ppy

module→get_usbCurrent()

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

cmd cpp vb cs java py php ts dnp ppy

module→get_userData()

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

cpp vb cs java py php ts ppy

module→get_userVar()

Returns the value previously stored in this attribute.

cmd cpp vb cs java py php ts dnp ppy

module→hasFunction(funcId)

Tests if the device includes a specific function.

cmd cpp vb cs java py php ts dnp ppy

module→isOnline()

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

cpp vb cs java py php ts dnp ppy

module→isOnline_async(callback, context)

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

module→isReadOnly()

Indicates whether changes to the module are prohibited or allowed.

cpp vb cs java py php ts dnp ppy cmd

module→load(msValidity)

Preloads the module cache with a specified validity duration.

cpp vb cs java py php ts ppy

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.

cmd cpp vb cs java py php ts dnp ppy

module→nextModule()

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

cpp vb cs java py php ts ppy

module→reboot(secBeforeReboot)

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

cmd cpp vb cs java py php ts dnp ppy

module→registerBeaconCallback(callback)

Register a callback function, to be called when the localization beacon of the module has been changed.

cpp vb cs java py php ts ppy

module→registerConfigChangeCallback(callback)

Register a callback function, to be called when a persistent settings in a device configuration has been changed (e.g.

cpp vb cs java py php ts ppy

module→registerLogCallback(callback)

Registers a device log callback function.

cpp vb cs java py php ts ppy

module→revertFromFlash()

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

cmd cpp vb cs java py php ts dnp ppy

module→saveToFlash()

Saves current settings in the nonvolatile memory of the module.

cmd cpp vb cs java py php ts dnp ppy

module→set_allSettings(settings)

Restores all the settings of the device.

cmd cpp vb cs java py php ts dnp ppy

module→set_allSettingsAndFiles(settings)

Restores all the settings and uploaded files to the module.

cmd cpp vb cs java py php ts dnp ppy

module→set_beacon(newval)

Turns on or off the module localization beacon.

cmd cpp vb cs java py php ts dnp ppy

module→set_logicalName(newval)

Changes the logical name of the module.

cpp vb cs java py php ts dnp ppy cmd

module→set_luminosity(newval)

Changes the luminosity of the module informative leds.

cmd cpp vb cs java py php ts dnp ppy

module→set_userData(data)

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

cpp vb cs java py php ts ppy

module→set_userVar(newval)

Stores a 32 bit value in the device RAM.

cmd cpp vb cs java py php ts dnp ppy

module→triggerConfigChangeCallback()

Triggers a configuration change callback, to check if they are supported or not.

cmd cpp vb cs java py php ts dnp ppy

module→triggerFirmwareUpdate(secBeforeReboot)

Schedules a module reboot into special firmware update mode.

cmd cpp vb cs java py php ts dnp ppy

module→updateFirmware(path)

Prepares a firmware update of the module.

cmd cpp vb cs java py php ts dnp ppy

module→updateFirmwareEx(path, force)

Prepares a firmware update of the module.

cmd cpp vb cs java py php ts dnp ppy

module→wait_async(callback, context)

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

ts

25.3. 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:

cpp	<code>#include "yocto_serialport.h"</code>
vb	<code>yocto_serialport.vb</code>
cs	<code>yocto_serialport.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YSerialPort;</code>
py	<code>from yocto_serialport import *</code>
php	<code>require_once('yocto_serialport.php');</code>
ts	<code>in HTML: import { YSerialPort } from '../dist/esm/yocto_serialport.js'; in Node.js: import { YSerialPort } from 'yoctolib-cjs/yocto_serialport.js';</code>
dnp	<code>import YoctoProxyAPI.YSerialPortProxy</code>
cp	<code>#include "yocto_serialport_proxy.h"</code>
ppy	<code>from yoctolib.yocto_serialport import *</code>
vi	<code>YSerialPort.vi</code>
ml	<code>import YoctoProxyAPI.YSerialPortProxy</code>

Global functions

`YSerialPort.FindSerialPort(func)`

Retrieves a serial port for a given identifier.

cpp vb cs java py php ts dnp ppy

`YSerialPort.FindSerialPortInContext(yctx, func)`

Retrieves a serial port for a given identifier in a YAPI context.

java ts ppy

`YSerialPort.FirstSerialPort()`

Starts the enumeration of serial ports currently accessible.

cpp vb cs java py php ts ppy

`YSerialPort.FirstSerialPortInContext(yctx)`

Starts the enumeration of serial ports currently accessible.

java ts ppy

`YSerialPort.GetSimilarFunctions()`

Enumerates all functions of type `SerialPort` available on the devices currently reachable by the library, and returns their unique hardware ID.

dnp

YSerialPort properties

`serialport→AdvertisedValue [read-only]`

Short string representing the current state of the function.

dnp

serialport→FriendlyName [read-only]

Global identifier of the function in the format `MODULE_NAME . FUNCTION_NAME`.

dnf

serialport→FunctionId [read-only]

Hardware identifier of the serial port, without reference to the module.

dnf

serialport→HardwareId [read-only]

Unique hardware identifier of the function in the form `SERIAL . FUNCTIONID`.

dnf

serialport→IsOnline [read-only]

Checks if the function is currently reachable.

dnf

serialport→JobMaxSize [read-only]

Maximum size allowed for job files.

dnf

serialport→JobMaxTask [read-only]

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

dnf

serialport→LogicalName [writable]

Logical name of the function.

dnf

serialport→Protocol [writable]

Type of protocol used over the serial line, as a string.

dnf

serialport→SerialMode [writable]

Serial port communication parameters, as a string such as "9600,8N1".

dnf

serialport→SerialNumber [read-only]

Serial number of the module, as set by the factory.

dnf

serialport→StartupJob [writable]

Job file to use when the device is powered on.

dnf

serialport→VoltageLevel [writable]

Voltage level used on the serial line.

dnf

YSerialPort methods

serialport→clearCache()

Invalidates the cache.

cpp

vb

cs

java

py

php

ts

ppy

serialport→describe()

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

`cpp` `vb` `cs` `java` `py` `php` `ts`

serialport→get_CTS()

Reads the level of the CTS line.

`cmd` `cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `ppy`

serialport→get_advertisedValue()

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

`cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `ppy` `cmd`

serialport→get_currentJob()

Returns the name of the job file currently in use.

`cmd` `cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `ppy`

serialport→get_errCount()

Returns the total number of communication errors detected since last reset.

`cmd` `cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `ppy`

serialport→get_errorMessage()

Returns the error message of the latest error with the serial port.

`cpp` `vb` `cs` `java` `py` `php` `ts` `ppy`

serialport→get_errorType()

Returns the numerical error code of the latest error with the serial port.

`cpp` `vb` `cs` `java` `py` `php` `ts` `ppy`

serialport→get_friendlyName()

Returns a global identifier of the serial port in the format `MODULE_NAME.FUNCTION_NAME`.

`cpp` `cs` `java` `py` `php` `ts` `dnp`

serialport→get_functionDescriptor()

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

`cpp` `vb` `cs` `java` `py` `php` `ts`

serialport→get_functionId()

Returns the hardware identifier of the serial port, without reference to the module.

`cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `ppy`

serialport→get_hardwareId()

Returns the unique hardware identifier of the serial port in the form `SERIAL.FUNCTIONID`.

`cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `ppy`

serialport→get_jobMaxSize()

Returns maximum size allowed for job files.

`cmd` `cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `ppy`

serialport→get_jobMaxTask()

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

`cmd` `cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `ppy`

serialport→get_lastMsg()

Returns the latest message fully received (for Line, Frame and Modbus protocols).

cmd cpp vb cs java py php ts dnp ppy

serialport→get_logicalName()

Returns the logical name of the serial port.

cpp vb cs java py php ts dnp ppy cmd

serialport→get_module()

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

cpp vb cs java py php ts dnp ppy

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.

cmd cpp vb cs java py php ts dnp ppy

serialport→get_rxCount()

Returns the total number of bytes received since last reset.

cmd cpp vb cs java py php ts dnp ppy

serialport→get_rxMsgCount()

Returns the total number of messages received since last reset.

cmd cpp vb cs java py php ts dnp ppy

serialport→get_serialMode()

Returns the serial port communication parameters, as a string such as "9600,8N1".

cmd cpp vb cs java py php ts dnp ppy

serialport→get_serialNumber()

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

cpp vb cs java py php ts dnp ppy cmd

serialport→get_startupJob()

Returns the job file to use when the device is powered on.

cmd cpp vb cs java py php ts dnp ppy

serialport→get_txCount()

Returns the total number of bytes transmitted since last reset.

cmd cpp vb cs java py php ts dnp ppy

serialport→get_txMsgCount()

Returns the total number of messages send since last reset.

cmd cpp vb cs java py php ts dnp ppy

serialport→get_userData()

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

cpp vb cs java py php ts ppy

serialport→get_voltageLevel()

Returns the voltage level used on the serial line.

cmd cpp vb cs java py php ts dnp ppy

serialport→isOnline()

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

cpp vb cs java py php ts dnp ppy

serialport→isOnline_async(callback, context)

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

serialport→isReadOnly()

Indicates whether changes to the function are prohibited or allowed.

cpp vb cs java py php ts dnp ppy cmd

serialport→load(msValidity)

Preloads the serial port cache with a specified validity duration.

cpp vb cs java py php ts ppy

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.

cpp vb cs java py php ts dnp ppy

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.

cmd cpp vb cs java py php ts dnp

serialport→modbusReadInputBits(slaveNo, pduAddr, nBits)

Reads one or more contiguous input bits (or discrete inputs) from a MODBUS serial device.

cmd cpp vb cs java py php ts dnp

serialport→modbusReadInputRegisters(slaveNo, pduAddr, nWords)

Reads one or more contiguous input registers (read-only registers) from a MODBUS serial device.

cmd cpp vb cs java py php ts dnp

serialport→modbusReadRegisters(slaveNo, pduAddr, nWords)

Reads one or more contiguous internal registers (holding registers) from a MODBUS serial device.

cmd cpp vb cs java py php ts dnp

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.

cmd cpp vb cs java py php ts dnp

serialport→modbusWriteBit(slaveNo, pduAddr, value)

Sets a single internal bit (or coil) on a MODBUS serial device.

cmd cpp vb cs java py php ts dnp ppy

serialport→modbusWriteBits(slaveNo, pduAddr, bits)

Sets several contiguous internal bits (or coils) on a MODBUS serial device.

cmd cpp vb cs java py php ts dnp ppy

serialport→modbusWriteRegister(slaveNo, pduAddr, value)

Sets a single internal register (or holding register) on a MODBUS serial device.

cmd cpp vb cs java py php ts dnp ppy

serialport→modbusWriteRegisters(slaveNo, pduAddr, values)

Sets several contiguous internal registers (or holding registers) on a MODBUS serial device.

cmd cpp vb cs java py php ts dnp ppy

serialport→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

cpp vb cs java py php ts dnp ppy cmd

serialport→nextSerialPort()

Continues the enumeration of serial ports started using `yFirstSerialPort()`.

cpp vb cs java py php ts ppy

serialport→queryHex(hexString, maxWait)

Sends a binary message to the serial port, and reads the reply, if any.

cmd cpp vb cs java py php ts dnp ppy

serialport→queryLine(query, maxWait)

Sends a text line query to the serial port, and reads the reply, if any.

cmd cpp vb cs java py php ts dnp ppy

serialport→queryMODBUS(slaveNo, pduBytes)

Sends a message to a specified MODBUS slave connected to the serial port, and reads the reply, if any.

cmd cpp vb cs java py php ts dnp

serialport→readArray(nChars)

Reads data from the receive buffer as a list of bytes, starting at current stream position.

cmd cpp vb cs java py php ts dnp

serialport→readBin(nChars)

Reads data from the receive buffer as a binary buffer, starting at current stream position.

cmd cpp vb cs java py php ts dnp ppy

serialport→readByte()

Reads one byte from the receive buffer, starting at current stream position.

cmd cpp vb cs java py php ts dnp ppy

serialport→readHex(nBytes)

Reads data from the receive buffer as a hexadecimal string, starting at current stream position.

cmd cpp vb cs java py php ts dnp ppy

serialport→readLine()

Reads a single line (or message) from the receive buffer, starting at current stream position.

cmd cpp vb cs java py php ts dnp ppy

serialport→readMessages(pattern, maxWait)

Searches for incoming messages in the serial port receive buffer matching a given pattern, starting at current position.

cmd cpp vb cs java py php ts dnp

serialport→readStr(nChars)

Reads data from the receive buffer as a string, starting at current stream position.

cmd cpp vb cs java py php ts dnp ppy

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.

cmd cpp vb cs java py php ts dnp ppy

serialport→read_seek(absPos)

Changes the current internal stream position to the specified value.

cmd cpp vb cs java py php ts dnp ppy

serialport→read_tell()

Returns the current absolute stream position pointer of the API object.

cmd cpp vb cs java py php ts dnp ppy

serialport→registerSnoopingCallback(callback)

Registers a callback function to be called each time that a message is sent or received by the serial port.

cpp vb cs java py php ts ppy

serialport→registerValueCallback(callback)

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

cpp vb cs java py php ts ppy

serialport→reset()

Clears the serial port buffer and resets counters to zero.

cmd cpp vb cs java py php ts dnp ppy

serialport→selectJob(jobfile)

Load and start processing the specified job file.

cmd cpp vb cs java py php ts dnp ppy

serialport→sendBreak(duration)

Emits a BREAK condition on the serial interface.

cmd cpp vb cs java py php ts dnp ppy

serialport→set_RTS(val)

Manually sets the state of the RTS line.

cmd cpp vb cs java py php ts dnp ppy

serialport→set_currentJob(newval)

Selects a job file to run immediately.

cmd cpp vb cs java py php ts dnp ppy

serialport→set_logicalName(newval)

Changes the logical name of the serial port.

cpp vb cs java py php ts dnp ppy cmd

serialport→set_protocol(newval)

Changes the type of protocol used over the serial line.

cmd cpp vb cs java py php ts dnp ppy

serialport→set_serialMode(newval)

Changes the serial port communication parameters, with a string such as "9600,8N1".

cmd cpp vb cs java py php ts dnp ppy

serialport→set_startupJob(newval)

Changes the job to use when the device is powered on.

cmd cpp vb cs java py php ts dnp ppy

serialport→set_userData(data)

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

cpp vb cs java py php ts ppy

serialport→**set_voltageLevel**(newval)

Changes the voltage type used on the serial line.

cmd cpp vb cs java py php ts dnp ppy

serialport→**snoopMessages**(maxWait)

Retrieves messages (both direction) in the serial port buffer, starting at current position.

cmd cpp vb cs java py php ts dnp

serialport→**snoopMessagesEx**(maxWait, maxMsg)

Retrieves messages (both direction) in the serial port buffer, starting at current position.

cmd cpp vb cs java py php ts dnp

serialport→**unmuteValueCallbacks**()

Re-enables the propagation of every new advertised value to the parent hub.

cpp vb cs java py php ts dnp ppy cmd

serialport→**uploadJob**(jobfile, jsonDef)

Saves the job definition string (JSON data) into a job file.

cmd cpp vb cs java py php ts dnp ppy

serialport→**wait_async**(callback, context)

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

ts

serialport→**writeArray**(byteList)

Sends a byte sequence (provided as a list of bytes) to the serial port.

cmd cpp vb cs java py php ts dnp ppy

serialport→**writeBin**(buff)

Sends a binary buffer to the serial port, as is.

cmd cpp vb cs java py php ts dnp ppy

serialport→**writeByte**(code)

Sends a single byte to the serial port.

cmd cpp vb cs java py php ts dnp ppy

serialport→**writeHex**(hexString)

Sends a byte sequence (provided as a hexadecimal string) to the serial port.

cmd cpp vb cs java py php ts dnp ppy

serialport→**writeLine**(text)

Sends an ASCII string to the serial port, followed by a line break (CR LF).

cmd cpp vb cs java py php ts dnp ppy

serialport→**writeMODBUS**(hexString)

Sends a MODBUS message (provided as a hexadecimal string) to the serial port.

cmd cpp vb cs java py php ts dnp ppy

serialport→**writeStr**(text)

Sends an ASCII string to the serial port, as is.

cmd cpp vb cs java py php ts dnp ppy

serialport→writeStxEtx(text)

Sends an ASCII string to the serial port, preceded with an STX code and followed by an ETX code.

cmd cpp vb cs java py php ts dnp ppy

25.4. Class YFiles

Filesystem control interface, available for instance in the Yocto-Color-V2, the Yocto-SPI, the YoctoHub-Ethernet or the YoctoHub-GSM-4G

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:

cpp	<code>#include "yocto_files.h"</code>
vb	<code>yocto_files.vb</code>
cs	<code>yocto_files.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YFiles;</code>
py	<code>from yocto_files import *</code>
php	<code>require_once('yocto_files.php');</code>
ts	<code>in HTML: import { YFiles } from '../dist/esm/yocto_files.js';</code> <code>in Node.js: import { YFiles } from 'yoctolib-cjs/yocto_files.js';</code>
dnf	<code>import YoctoProxyAPI.YFilesProxy</code>
cp	<code>#include "yocto_files_proxy.h"</code>
ppy	<code>from yoctolib.yocto_files import *</code>
vi	<code>YFiles.vi</code>
m1	<code>import YoctoProxyAPI.YFilesProxy</code>

Global functions

YFiles.FindFiles(func)

Retrieves a filesystem for a given identifier.

cpp vb cs java py php ts dnf ppy

YFiles.FindFilesInContext(yctx, func)

Retrieves a filesystem for a given identifier in a YAPI context.

java ts ppy

YFiles.FirstFiles()

Starts the enumeration of filesystems currently accessible.

cpp vb cs java py php ts ppy

YFiles.FirstFilesInContext(yctx)

Starts the enumeration of filesystems currently accessible.

java ts ppy

YFiles.GetSimilarFunctions()

Enumerates all functions of type Files available on the devices currently reachable by the library, and returns their unique hardware ID.

dnf

YFiles properties

files→AdvertisedValue [read-only]

Short string representing the current state of the function.

dnf

files→FilesCount [read-only]

Number of files currently loaded in the filesystem.

dnf

files→FriendlyName *[read-only]*

Global identifier of the function in the format `MODULE_NAME . FUNCTION_NAME`.

dnf

files→FunctionId *[read-only]*

Hardware identifier of the filesystem, without reference to the module.

dnf

files→HardwareId *[read-only]*

Unique hardware identifier of the function in the form `SERIAL . FUNCTIONID`.

dnf

files→IsOnline *[read-only]*

Checks if the function is currently reachable.

dnf

files→LogicalName *[writable]*

Logical name of the function.

dnf

files→SerialNumber *[read-only]*

Serial number of the module, as set by the factory.

dnf

YFiles methods

files→clearCache()

Invalidates the cache.

cpp vb cs java py php ts ppy

files→describe()

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

cpp vb cs java py php ts

files→download(pathname)

Downloads the requested file and returns a binary buffer with its content.

cmd cpp vb cs java py php ts dnf ppy

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.

cmd cpp vb cs java py php ts dnf ppy

files→format_fs()

Reinitialize the filesystem to its clean, unfragmented, empty state.

cmd cpp vb cs java py php ts dnf ppy

files→get_advertisedValue()

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

cpp vb cs java py php ts dnp ppy cmd

files→get_errorMessage()

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

cpp vb cs java py php ts ppy

files→get_errorType()

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

cpp vb cs java py php ts ppy

files→get_filesCount()

Returns the number of files currently loaded in the filesystem.

cmd cpp vb cs java py php ts dnp ppy

files→get_freeSpace()

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

cmd cpp vb cs java py php ts dnp ppy

files→get_friendlyName()

Returns a global identifier of the filesystem in the format `MODULE_NAME . FUNCTION_NAME`.

cpp cs java py php ts dnp

files→get_functionDescriptor()

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

cpp vb cs java py php ts

files→get_functionId()

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

cpp vb cs java py php ts dnp ppy

files→get_hardwareId()

Returns the unique hardware identifier of the filesystem in the form `SERIAL . FUNCTIONID`.

cpp vb cs java py php ts dnp ppy

files→get_list(pattern)

Returns a list of `YFileRecord` objects that describe files currently loaded in the filesystem.

cmd cpp vb cs java py php ts dnp

files→get_logicalName()

Returns the logical name of the filesystem.

cpp vb cs java py php ts dnp ppy cmd

files→get_module()

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

cpp vb cs java py php ts dnp ppy

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.

cpp vb cs java py php ts dnp ppy cmd

files→get_userData()

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

cpp vb cs java py php ts ppy

files→isOnline()

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

cpp vb cs java py php ts dnp ppy

files→isOnline_async(callback, context)

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

files→isReadOnly()

Indicates whether changes to the function are prohibited or allowed.

cpp vb cs java py php ts dnp ppy cmd

files→load(msValidity)

Preloads the filesystem cache with a specified validity duration.

cpp vb cs java py php ts ppy

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.

cpp vb cs java py php ts dnp ppy

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.

cpp vb cs java py php ts dnp ppy cmd

files→nextFiles()

Continues the enumeration of filesystems started using `yFirstFiles()`.

cpp vb cs java py php ts ppy

files→registerValueCallback(callback)

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

cpp vb cs java py php ts ppy

files→remove(pathname)

Deletes a file, given by its full path name, from the filesystem.

cmd cpp vb cs java py php ts dnp ppy

files→set_logicalName(newval)

Changes the logical name of the filesystem.

cpp vb cs java py php ts dnp ppy cmd

files→set_userData(data)

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

cpp vb cs java py php ts ppy

files→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

cpp vb cs java py php ts dnp ppy cmd

files→upload(pathname, content)

Uploads a file to the filesystem, to the specified full path name.

cmd cpp vb cs java py php ts dnp ppy

files→**wait_async(callback, context)**

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

ts

25.5. Class YGenericSensor

GenericSensor control interface, available for instance in the Yocto-0-10V-Rx, the Yocto-4-20mA-Rx, the Yocto-Bridge 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:

cpp	<code>#include "yocto_genericsensor.h"</code>
vb	<code>yocto_genericsensor.vb</code>
cs	<code>yocto_genericsensor.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YGenericSensor;</code>
py	<code>from yocto_genericsensor import *</code>
php	<code>require_once('yocto_genericsensor.php');</code>
ts	<code>in HTML: import { YGenericSensor } from '../dist/esm/yocto_genericsensor.js';</code> <code>in Node.js: import { YGenericSensor } from 'yoctolib-cjs/yocto_genericsensor.js';</code>
dnf	<code>import YoctoProxyAPI.YGenericSensorProxy</code>
cp	<code>#include "yocto_genericsensor_proxy.h"</code>
ppy	<code>from yoctolib.yocto_genericsensor import *</code>
vi	<code>YGenericSensor.vi</code>
ml	<code>import YoctoProxyAPI.YGenericSensorProxy</code>

Global functions

YGenericSensor.FindGenericSensor(func)

Retrieves a generic sensor for a given identifier.

cpp vb cs java py php ts dnf ppy

YGenericSensor.FindGenericSensorInContext(yctx, func)

Retrieves a generic sensor for a given identifier in a YAPI context.

java ts ppy

YGenericSensor.FirstGenericSensor()

Starts the enumeration of generic sensors currently accessible.

cpp vb cs java py php ts ppy

YGenericSensor.FirstGenericSensorInContext(yctx)

Starts the enumeration of generic sensors currently accessible.

java ts ppy

YGenericSensor.GetSimilarFunctions()

Enumerates all functions of type GenericSensor available on the devices currently reachable by the library, and returns their unique hardware ID.

dnf

YGenericSensor properties

genericsensor→AdvMode [writable]

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

dnf

genericsensor→AdvertisedValue [read-only]

Short string representing the current state of the function.

dnf

genericsensor→Enabled [writable]

Activation state of this input.

dnf

genericsensor→FriendlyName [read-only]

Global identifier of the function in the format `MODULE_NAME . FUNCTION_NAME`.

dnf

genericsensor→FunctionId [read-only]

Hardware identifier of the sensor, without reference to the module.

dnf

genericsensor→HardwareId [read-only]

Unique hardware identifier of the function in the form `SERIAL . FUNCTIONID`.

dnf

genericsensor→IsOnline [read-only]

Checks if the function is currently reachable.

dnf

genericsensor→LogFrequency [writable]

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

dnf

genericsensor→LogicalName [writable]

Logical name of the function.

dnf

genericsensor→ReportFrequency [writable]

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

dnf

genericsensor→Resolution [writable]

Resolution of the measured values.

dnf

genericsensor→SerialNumber [read-only]

Serial number of the module, as set by the factory.

dnf

genericsensor→SignalBias [writable]

Electric signal bias for zero shift adjustment.

dnf

genericsensor→SignalRange [writable]

Input signal range used by the sensor.

dnf

genericsensor→SignalSampling [writable]

Electric signal sampling method to use.

dnf

genericSensor→SignalUnit [read-only]

Measuring unit of the electrical signal used by the sensor.

dnf

genericSensor→ValueRange [writable]

Physical value range measured by the sensor.

dnf

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.

cmd cpp vb cs java py php ts dnf

genericSensor→clearCache()

Invalidates the cache.

cpp vb cs java py php ts ppy

genericSensor→describe()

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

cpp vb cs java py php ts

genericSensor→get_advMode()

Returns the measuring mode used for the advertised value pushed to the parent hub.

cmd cpp vb cs java py php ts dnf

genericSensor→get_advertisedValue()

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

cpp vb cs java py php ts dnf ppy cmd

genericSensor→get_currentRawValue()

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

cmd cpp vb cs java py php ts dnf

genericSensor→get_currentValue()

Returns the current measured value.

cmd cpp vb cs java py php ts dnf

genericSensor→get_dataLogger()

Returns the `YDataLogger` object of the device hosting the sensor.

cpp vb cs java py php ts dnf

genericSensor→get_enabled()

Returns the activation state of this input.

cmd cpp vb cs java py php ts dnf ppy

genericSensor→get_errorMessage()

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

cpp vb cs java py php ts ppy

genericsensor→get_errorType()

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

cpp vb cs java py php ts ppy

genericsensor→get_friendlyName()

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

cpp cs java py php ts dnp

genericsensor→get_functionDescriptor()

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

cpp vb cs java py php ts

genericsensor→get_functionId()

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

cpp vb cs java py php ts dnp ppy

genericsensor→get_hardwareId()

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

cpp vb cs java py php ts dnp ppy

genericsensor→get_highestValue()

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

cmd cpp vb cs java py php ts dnp

genericsensor→get_logFrequency()

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

cmd cpp vb cs java py php ts dnp

genericsensor→get_logicalName()

Returns the logical name of the generic sensor.

cpp vb cs java py php ts dnp ppy cmd

genericsensor→get_lowestValue()

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

cmd cpp vb cs java py php ts dnp

genericsensor→get_module()

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

cpp vb cs java py php ts dnp ppy

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.

cmd cpp vb cs java py php ts dnp

genericsensor→get_reportFrequency()

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

cmd cpp vb cs java py php ts dnp

genericsensor→get_resolution()

Returns the resolution of the measured values.

cmd cpp vb cs java py php ts dnp

genericSensor→get_sensorState()

Returns the sensor 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.

cmd cpp vb cs java py php ts dnp

genericSensor→get_serialNumber()

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

cpp vb cs java py php ts dnp ppy cmd

genericSensor→get_signalBias()

Returns the electric signal bias for zero shift adjustment.

cmd cpp vb cs java py php ts dnp ppy

genericSensor→get_signalRange()

Returns the input signal range used by the sensor.

cmd cpp vb cs java py php ts dnp ppy

genericSensor→get_signalSampling()

Returns the electric signal sampling method to use.

cmd cpp vb cs java py php ts dnp ppy

genericSensor→get_signalUnit()

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

cmd cpp vb cs java py php ts dnp ppy

genericSensor→get_signalValue()

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

cmd cpp vb cs java py php ts dnp ppy

genericSensor→get_unit()

Returns the measuring unit for the measure.

cmd cpp vb cs java py php ts dnp

genericSensor→get_userData()

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

cpp vb cs java py php ts ppy

genericSensor→get_valueRange()

Returns the physical value range measured by the sensor.

cmd cpp vb cs java py php ts dnp ppy

genericSensor→isOnline()

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

cpp vb cs java py php ts dnp ppy

genericSensor→isOnline_async(callback, context)

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

genericSensor→isReadOnly()

Indicates whether changes to the function are prohibited or allowed.

cpp vb cs java py php ts dnp ppy cmd

genericsensor→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

cmd

genericsensor→load(msValidity)

Preloads the generic sensor cache with a specified validity duration.

cpp vb cs java py php ts ppy

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.

cpp vb cs java py php ts dnp ppy

genericsensor→loadCalibrationPoints(rawValues, refValues)

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

cmd cpp vb cs java py php ts

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.

cpp vb cs java py php ts dnp ppy cmd

genericsensor→nextGenericSensor()

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

cpp vb cs java py php ts ppy

genericsensor→registerTimedReportCallback(callback)

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

cpp vb cs java py php ts ppy

genericsensor→registerValueCallback(callback)

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

cpp vb cs java py php ts ppy

genericsensor→set_advMode(newval)

Changes the measuring mode used for the advertised value pushed to the parent hub.

cmd cpp vb cs java py php ts dnp

genericsensor→set_enabled(newval)

Changes the activation state of this input.

cmd cpp vb cs java py php ts dnp ppy

genericsensor→set_highestValue(newval)

Changes the recorded maximal value observed.

cmd cpp vb cs java py php ts dnp

genericsensor→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

cmd cpp vb cs java py php ts dnp

genericsensor→set_logicalName(newval)

Changes the logical name of the generic sensor.

cpp vb cs java py php ts dnp ppy cmd

genericsensor→set_lowestValue(newval)

Changes the recorded minimal value observed.

cmd cpp vb cs java py php ts dnp

genericsensor→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

cmd cpp vb cs java py php ts dnp

genericsensor→set_resolution(newval)

Modifies the resolution of the measured physical values.

cmd cpp vb cs java py php ts dnp

genericsensor→set_signalBias(newval)

Changes the electric signal bias for zero shift adjustment.

cmd cpp vb cs java py php ts dnp ppy

genericsensor→set_signalRange(newval)

Changes the input signal range used by the sensor.

cmd cpp vb cs java py php ts dnp ppy

genericsensor→set_signalSampling(newval)

Changes the electric signal sampling method to use.

cmd cpp vb cs java py php ts dnp ppy

genericsensor→set_unit(newval)

Changes the measuring unit for the measured value.

cmd cpp vb cs java py php ts dnp ppy

genericsensor→set_userData(data)

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

cpp vb cs java py php ts ppy

genericsensor→set_valueRange(newval)

Changes the output value range, corresponding to the physical value measured by the sensor.

cmd cpp vb cs java py php ts dnp ppy

genericsensor→startDataLogger()

Starts the data logger on the device.

cmd cpp vb cs java py php ts dnp

genericsensor→stopDataLogger()

Stops the datalogger on the device.

cmd cpp vb cs java py php ts dnp

genericsensor→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

cpp vb cs java py php ts dnp ppy cmd

genericsensor→wait_async(callback, context)

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

ts

genericsensor→zeroAdjust()

Adjusts the signal bias so that the current signal value is need precisely as zero.

cmd

cpp

vb

cs

java

py

php

ts

dnp

ppy

25.6. 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 retrieval is done at sensor objects level.

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

cpp	<code>#include "yocto_module.h"</code>
vb	<code>yocto_module.vb</code>
cs	<code>yocto_module.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YDataLogger;</code>
py	<code>from yocto_module import *</code>
php	<code>require_once('yocto_module.php');</code>
ts	<code>in HTML: import { YDataLogger } from '../dist/esm/yocto_module.js'; in Node.js: import { YDataLogger } from 'yoctolib-cjs/yocto_module.js';</code>
dnf	<code>import YoctoProxyAPI.YDataLoggerProxy</code>
cp	<code>#include "yocto_module_proxy.h"</code>
ppy	<code>from yoctolib.yocto_module import *</code>
vi	<code>YDataLogger.vi</code>
ml	<code>import YoctoProxyAPI.YDataLoggerProxy</code>

Global functions

YDataLogger.FindDataLogger(func)

Retrieves a data logger for a given identifier.

cpp vb cs java py php ts dnf

YDataLogger.FindDataLoggerInContext(yctx, func)

Retrieves a data logger for a given identifier in a YAPI context.

java ts

YDataLogger.FirstDataLogger()

Starts the enumeration of data loggers currently accessible.

cpp vb cs java py php ts

YDataLogger.FirstDataLoggerInContext(yctx)

Starts the enumeration of data loggers currently accessible.

java ts

YDataLogger.GetSimilarFunctions()

Enumerates all functions of type DataLogger available on the devices currently reachable by the library, and returns their unique hardware ID.

dnf

YDataLogger properties

datalogger→AdvertisedValue [read-only]

Short string representing the current state of the function.

dnf

datalogger→AutoStart [writable]

Default activation state of the data logger on power up.

dnf

datalogger→BeaconDriven [writable]

True if the data logger is synchronised with the localization beacon.

dnf

datalogger→FriendlyName [read-only]

Global identifier of the function in the format `MODULE_NAME . FUNCTION_NAME`.

dnf

datalogger→FunctionId [read-only]

Hardware identifier of the data logger, without reference to the module.

dnf

datalogger→HardwareId [read-only]

Unique hardware identifier of the function in the form `SERIAL . FUNCTIONID`.

dnf

datalogger→IsOnline [read-only]

Checks if the function is currently reachable.

dnf

datalogger→LogicalName [writable]

Logical name of the function.

dnf

datalogger→Recording [writable]

Current activation state of the data logger.

dnf

datalogger→SerialNumber [read-only]

Serial number of the module, as set by the factory.

dnf

YDataLogger methods

datalogger→clearCache()

Invalidates the cache.

cpp vb cs java py php ts ppy

datalogger→describe()

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

cpp vb cs java py php ts

datalogger→forgetAllDataStreams()

Clears the data logger memory and discards all recorded data streams.

cmd cpp vb cs java py php ts dnf

datalogger→get_advertisedValue()

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

cpp vb cs java py php ts dnf ppy cmd

datalogger→get_autoStart()

Returns the default activation state of the data logger on power up.

cmd cpp vb cs java py php ts dnp

datalogger→get_beaconDriven()

Returns true if the data logger is synchronised with the localization beacon.

cmd cpp vb cs java py php ts dnp

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.

cmd cpp vb cs java py php ts dnp

datalogger→get_dataSets()

Returns a list of YDataSet objects that can be used to retrieve all measures stored by the data logger.

cmd cpp vb cs java py php ts dnp

datalogger→get_errorMessage()

Returns the error message of the latest error with the data logger.

cpp vb cs java py php ts ppy

datalogger→get_errorType()

Returns the numerical error code of the latest error with the data logger.

cpp vb cs java py php ts ppy

datalogger→get_friendlyName()

Returns a global identifier of the data logger in the format `MODULE_NAME . FUNCTION_NAME`.

cpp cs java py php ts dnp

datalogger→get_functionDescriptor()

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

cpp vb cs java py php ts

datalogger→get_functionId()

Returns the hardware identifier of the data logger, without reference to the module.

cpp vb cs java py php ts dnp ppy

datalogger→get_hardwareId()

Returns the unique hardware identifier of the data logger in the form `SERIAL . FUNCTIONID`.

cpp vb cs java py php ts dnp ppy

datalogger→get_logicalName()

Returns the logical name of the data logger.

cpp vb cs java py php ts dnp ppy cmd

datalogger→get_module()

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

cpp vb cs java py php ts dnp ppy

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.

cmd cpp vb cs java py php ts dnp

datalogger→get_serialNumber()

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

cpp vb cs java py php ts dnp ppy cmd

datalogger→get_timeUTC()

Returns the Unix timestamp for current UTC time, if known.

cmd cpp vb cs java py php ts dnp

datalogger→get_usage()

Returns the percentage of datalogger memory in use.

cmd cpp vb cs java py php ts dnp

datalogger→get_userData()Returns the value of the userData attribute, as previously stored using method `set_userData`.

cpp vb cs java py php ts ppy

datalogger→isOnline()

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

cpp vb cs java py php ts dnp ppy

datalogger→isOnline_async(callback, context)

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

datalogger→isReadOnly()

Indicates whether changes to the function are prohibited or allowed.

cpp vb cs java py php ts dnp ppy cmd

datalogger→load(msValidity)

Preloads the data logger cache with a specified validity duration.

cpp vb cs java py php ts ppy

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.

cpp vb cs java py php ts dnp ppy

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.

cpp vb cs java py php ts dnp ppy cmd

datalogger→nextDataLogger()Continues the enumeration of data loggers started using `yFirstDataLogger()`.

cpp vb cs java py php ts

datalogger→registerValueCallback(callback)

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

cpp vb cs java py php ts ppy

datalogger→set_autoStart(newval)

Changes the default activation state of the data logger on power up.

cmd cpp vb cs java py php ts dnp

datalogger→set_beaconDriven(newval)

Changes the type of synchronisation of the data logger.

cmd cpp vb cs java py php ts dnp

datalogger→set_logicalName(newval)

Changes the logical name of the data logger.

cpp vb cs java py php ts dnp ppy cmd

datalogger→set_recording(newval)

Changes the activation state of the data logger to start/stop recording data.

cmd cpp vb cs java py php ts dnp

datalogger→set_timeUTC(newval)

Changes the current UTC time reference used for recorded data.

cmd cpp vb cs java py php ts dnp

datalogger→set_userData(data)

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

cpp vb cs java py php ts ppy

datalogger→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

cpp vb cs java py php ts dnp ppy cmd

datalogger→wait_async(callback, context)

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

ts

25.7. 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:

cpp	<code>#include "yocto_module.h"</code>
vb	<code>yocto_module.vb</code>
cs	<code>yocto_module.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YDataSet;</code>
py	<code>from yocto_module import *</code>
php	<code>require_once('yocto_module.php');</code>
ts	<code>in HTML: import { YDataSet } from '../dist/esm/yocto_module.js'; in Node.js: import { YDataSet } from 'yoctolib-cjs/yocto_module.js';</code>
dnf	<code>import YoctoProxyAPI.YDataSetProxy</code>
cp	<code>#include "yocto_module_proxy.h"</code>
ppy	<code>from yoctolib.yocto_module import *</code>
ml	<code>import YoctoProxyAPI.YDataSetProxy</code>

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.

cpp vb cs java py php ts dnf

dataset→get_functionId()

Returns the hardware identifier of the function that performed the measure, without reference to the module.

cpp vb cs java py php ts dnf

dataset→get_hardwareId()

Returns the unique hardware identifier of the function who performed the measures, in the form SERIAL.FUNCTIONID.

cpp vb cs java py php ts dnf

dataset→get_measures()

Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

cpp vb cs java py php ts dnp

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()`.

cpp vb cs java py php ts dnp

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 be retrieved in this `YDataSet`, as a list of `YMeasure` objects.

cpp vb cs java py php ts dnp

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.

cpp vb cs java py php ts dnp

dataset→get_startTimeUTC()

Returns the start time of the dataset, relative to the Jan 1, 1970.

cpp vb cs java py php ts dnp

dataset→get_summary()

Returns an `YMeasure` object which summarizes the whole `YDataSet`.

cpp vb cs java py php ts dnp

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.

[cpp](#) [vb](#) [cs](#) [java](#) [py](#) [php](#) [ts](#) [dnp](#)

dataset→loadMore()

Loads the next block of measures from the dataLogger, and updates the progress indicator.

[cpp](#) [vb](#) [cs](#) [java](#) [py](#) [php](#) [ts](#) [dnp](#)

dataset→loadMore_async(callback, context)

Loads the next block of measures from the dataLogger asynchronously.

25.8. 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:

cpp	<code>#include "yocto_module.h"</code>
vb	<code>yocto_module.vb</code>
cs	<code>yocto_module.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YMeasure;</code>
py	<code>from yocto_module import *</code>
php	<code>require_once('yocto_module.php');</code>
ts	<code>in HTML: import { YMeasure } from '../dist/esm/yocto_module.js';</code> <code>in Node.js: import { YMeasure } from 'yoctolib-cjs/yocto_module.js';</code>
ppy	<code>from yoctolib.yocto_module import *</code>

YMeasure methods

`measure→get_averageValue()`

Returns the average value observed during the time interval covered by this measure.

cpp vb cs java py php ts ppy

`measure→get_endTimeUTC()`

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

cpp vb cs java py php ts ppy

`measure→get_maxValue()`

Returns the largest value observed during the time interval covered by this measure.

cpp vb cs java py php ts ppy

`measure→get_minValue()`

Returns the smallest value observed during the time interval covered by this measure.

cpp vb cs java py php ts ppy

`measure→get_startTimeUTC()`

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

cpp vb cs java py php ts ppy

26. Troubleshooting

26.1. 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.

26.2. 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.

26.3. Linux and USB

To work correctly under Linux, the library needs to have write access to all the Yoctopuce USB peripherals. However, by default under Linux, USB privileges of the non-root users are limited to read access. To avoid having to run the library as root, you need to create a new *udev* rule to authorize one or several users to have write access to the Yoctopuce peripherals.

To add a new *udev* rule to your installation, you must add a file with a name following the "`##-arbitraryName.rules`" format, in the `/etc/udev/rules.d` directory. When the system is starting, *udev* reads all the files with a `.rules` extension in this directory, respecting the alphabetical order (for example, the `51-custom.rules` file is interpreted AFTER the `50-udev-default.rules` file).

The `50-udev-default` file contains the system default *udev* rules. To modify the default behavior, you therefore need to create a file with a name that starts with a number larger than 50, that will override the system default rules. Note that to add a rule, you need a root access on the system.

In the `udev_conf` directory of the VirtualHub for Linux³ archive, there are two rule examples which you can use as a basis.

¹ see: http://www.yoctopuce.com/EN/blog_by_categories/for-the-beginners

² see: <http://www.yoctopuce.com/EN/article/about-programming-examples>

³ <http://www.yoctopuce.com/FR/virtualhub.php>

Example 1: 51-yoctopuce.rules

This rule provides all the users with read and write access to the Yoctopuce USB devices. Access rights for all other devices 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"
```

26.4. ARM Platforms: HF and EL

There are two main flavors of executable on ARM: HF (Hard Float) binaries, and EL (EABI Little Endian) binaries. These two families are not compatible at all. The compatibility of a given ARM platform with one of these two families depends on the hardware and on the OS build. ArmHL and ArmEL compatibility problems are quite difficult to detect. Most of the time, the OS itself is unable to make a difference between an HF and an EL executable and will return meaningless messages when you try to use the wrong type of binary.

All pre-compiled Yoctopuce binaries are provided in both formats, as two separate ArmHF et ArmEL executables. If you do not know what family your ARM platform belongs to, just try one executable from each family.

26.5. Powered module but invisible for the OS

If your Yocto-RS232-C 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.

26.6. 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 VirtualHub and the network mode ⁴.

26.7. Disconnections, erratic behavior

If your Yocto-RS232-C 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>

26.8. After a failed firmware update, the device stopped working

If a firmware update of your Yocto-RS232-C fails, it is possible that the module is no longer working. If this is the case, plug in your module while holding down the Yocto-Button. The Yocto-LED should light up brightly and remain steady. Release the button. Your Yocto-RS232-C should then appear at the bottom of the VirtualHub user interface as a module waiting to be flashed. This operation also reverts the module to its factory configuration.

26.9. Registering VirtualHub disconnects another instance

If, when performing a call to `RegisterHub()` with a VirtualHub address, another previously registered VirtualHub disconnects, make sure the machine running these VirtualHubs do not 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 numbers are created on the fly based the *hostname* of the machine running VirtualHub.

26.10. 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, a 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()` waits until there is no more command to send before freeing the API resources and returning.

26.11. Damaged device

Yoctopuce strives to reduce the production of electronic waste. If you believe that your Yocto-RS232-C 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-RS232-C, 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.



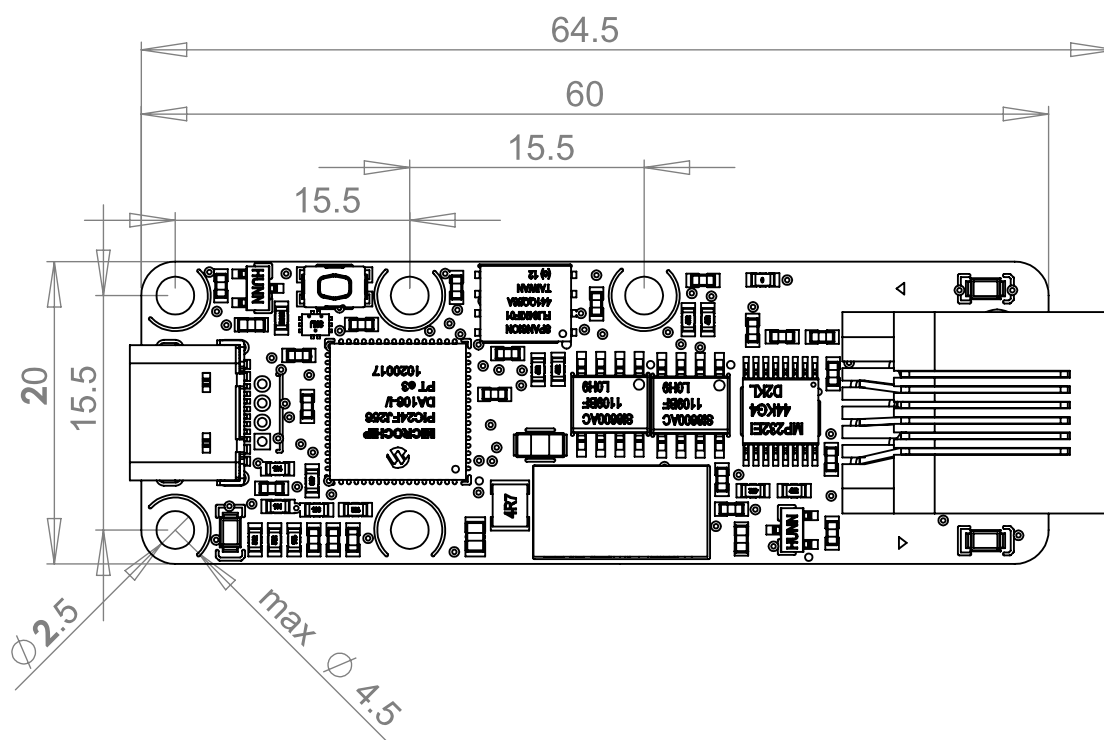
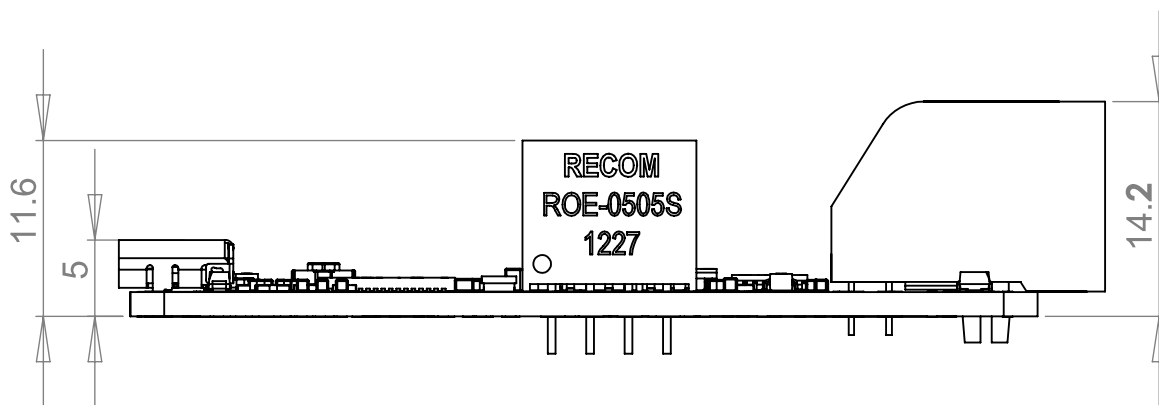
27. Characteristics

You can find below a summary of the main technical characteristics of your Yocto-RS232-C module.

Product ID	RS232M1C
Hardware release [†]	Rev. B
USB connector	USB-C
Width	20 mm
Length	60 mm
Weight	36 g
Chipset	MAX3232
Max frequency	115200 bps
Buffer	16 KB
Protection class, according to IEC 61140	class III
USB isolation, dielectric strength	1 kV
Normal operating temperature	5...40 °C
Extended operating temperature [‡]	-25...85 °C
RoHS compliance	RoHS III (2011/65/UE+2015/863)
USB Vendor ID	0x24E0
USB Device ID	0x00CE
Suggested enclosure	YoctoBox-Long-Thick-Black-RS232
Harmonized tariff code	9032.9000
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-RS232-C

A4

Scale
2:1
Echelle