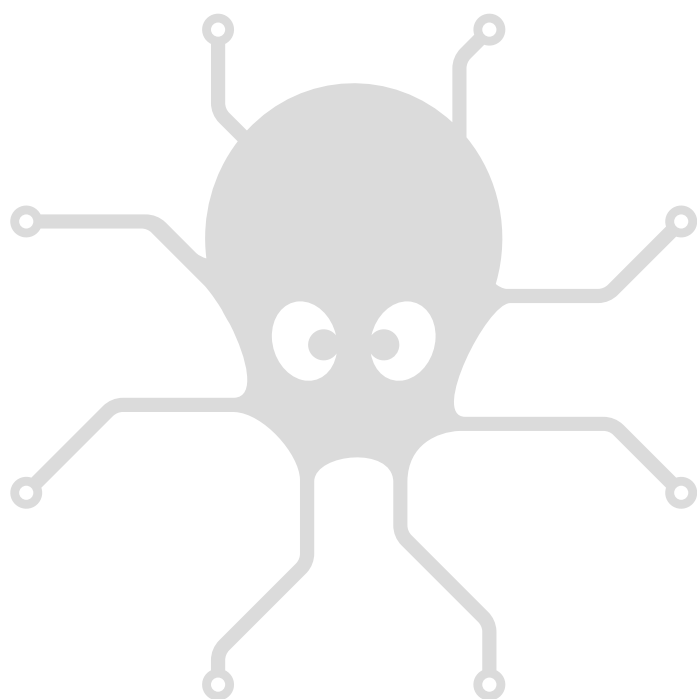




Yocto-Watt

User's guide

Table of contents

1. Introduction	1
1.1. <i>Safety Information</i>	2
1.2. <i>Electrocution Prevention Advice</i>	3
1.3. <i>Environmental conditions</i>	4
2. Presentation	5
2.1. <i>Common elements</i>	5
2.2. <i>Specific elements</i>	7
2.3. <i>Safety isolation</i>	8
2.4. <i>Optional accessories</i>	9
3. Direct current vs. alternating current	11
3.1. <i>Measure of the voltage</i>	11
3.2. <i>Measure of the current</i>	12
3.3. <i>Measure of the power</i>	12
4. First steps	15
4.1. <i>Prerequisites</i>	15
4.2. <i>Testing USB connectivity</i>	16
4.3. <i>Localization</i>	17
4.4. <i>Test of the module</i>	17
4.5. <i>Configuration</i>	17
5. Assembly and connections	19
5.1. <i>Fixing</i>	19
5.2. <i>USB power distribution</i>	19
6. Programming, general concepts	21
6.1. <i>Programming paradigm</i>	21
6.2. <i>The Yocto-Watt module</i>	23
6.3. <i>Module</i>	24
6.4. <i>Power</i>	25
6.5. <i>Voltage</i>	27
6.6. <i>Current</i>	28

6.7. DataLogger	29
6.8. What interface: Native, DLL or Service ?	30
6.9. Programming, where to start?	32
7. Using the Yocto-Watt in command line	35
7.1. Installing	35
7.2. Use: general description	35
7.3. Control of the Power function	36
7.4. Control of the module part	36
7.5. Limitations	37
8. Using the Yocto-Watt with Python	39
8.1. Source files	39
8.2. Dynamic library	39
8.3. Control of the Power function	39
8.4. Control of the module part	41
8.5. Error handling	43
9. Using Yocto-Watt with C++	45
9.1. Control of the Power function	45
9.2. Control of the module part	47
9.3. Error handling	50
9.4. Integration variants for the C++ Yoctopuce library	50
10. Using Yocto-Watt with C#	53
10.1. Installation	53
10.2. Using the Yoctopuce API in a Visual C# project	53
10.3. Control of the Power function	54
10.4. Control of the module part	56
10.5. Error handling	58
11. Using the Yocto-Watt with LabVIEW	61
11.1. Architecture	61
11.2. Compatibility	62
11.3. Installation	62
11.4. Presentation of Yoctopuce VIs	67
11.5. Functioning and use of VIs	70
11.6. Using	72
11.7. Managing the data logger	74
11.8. Function list	75
11.9. A word on performances	76
11.10. A full example of a LabVIEW program	76
11.11. Differences from other Yoctopuce APIs	77
12. Using the Yocto-Watt with Java	79
12.1. Getting ready	79
12.2. Control of the Power function	79
12.3. Control of the module part	81
12.4. Error handling	83
13. Using the Yocto-Watt with Android	85
13.1. Native access and VirtualHub	85

13.2. Getting ready	85
13.3. Compatibility	85
13.4. Activating the USB port under Android	86
13.5. Control of the Power function	87
13.6. Control of the module part	90
13.7. Error handling	95
14. Using Yocto-Watt with TypeScript	97
14.1. Using the Yoctopuce library for TypeScript	98
14.2. Refresher on asynchronous I/O in JavaScript	98
14.3. Control of the Power function	99
14.4. Control of the module part	102
14.5. Error handling	104
15. Using Yocto-Watt with JavaScript / EcmaScript	105
15.1. Blocking I/O versus Asynchronous I/O in JavaScript	105
15.2. Using Yoctopuce library for JavaScript / EcmaScript 2017	106
15.3. Control of the Power function	108
15.4. Control of the module part	111
15.5. Error handling	113
16. Using Yocto-Watt with PHP	115
16.1. Getting ready	115
16.2. Control of the Power function	115
16.3. Control of the module part	117
16.4. HTTP callback API and NAT filters	120
16.5. Error handling	123
17. Using Yocto-Watt with Visual Basic .NET	125
17.1. Installation	125
17.2. Using the Yoctopuce API in a Visual Basic project	125
17.3. Control of the Power function	126
17.4. Control of the module part	128
17.5. Error handling	130
18. Using Yocto-Watt with Delphi	131
18.1. Preparation	131
18.2. Control of the Power function	131
18.3. Control of the module part	133
18.4. Error handling	136
19. Using the Yocto-Watt with Universal Windows Platform	137
19.1. Blocking and asynchronous functions	137
19.2. Installation	138
19.3. Using the Yoctopuce API in a Visual Studio project	138
19.4. Control of the Power function	139
19.5. A real example	140
19.6. Control of the module part	140
19.7. Error handling	143
20. Using Yocto-Watt with Objective-C	145
20.1. Control of the Power function	145

20.2. Control of the module part	147
20.3. Error handling	149
21. Using with unsupported languages	151
21.1. Command line	151
21.2. .NET Assembly	151
21.3. VirtualHub and HTTP GET	153
21.4. Using dynamic libraries	155
21.5. Porting the high level library	158
22. Advanced programming	159
22.1. Event programming	159
22.2. The data logger	162
22.3. Sensor calibration	164
23. Firmware Update	169
23.1. The VirtualHub or the YoctoHub	169
23.2. The command line library	169
23.3. The Android application Yocto-Firmware	169
23.4. Updating the firmware with the programming library	170
23.5. The "update" mode	172
24. High-level API Reference	173
24.1. Class YAPI	174
24.2. Class YModule	215
24.3. Class YPower	292
24.4. Class YVoltage	370
24.5. Class YCurrent	444
24.6. Class YDataLogger	518
24.7. Class YDataSet	577
24.8. Class YMeasure	610
25. Troubleshooting	617
25.1. Where to start?	617
25.2. Programming examples don't seem to work	617
25.3. Linux and USB	617
25.4. ARM Platforms: HF and EL	618
25.5. Powered module but invisible for the OS	618
25.6. Another process named xxx is already using yAPI	618
25.7. Disconnections, erratic behavior	618
25.8.	619
25.9. Dropped commands	619
25.10. Damaged device	619
26. Characteristics	621

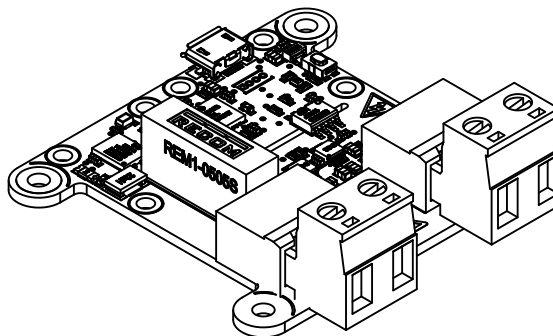
1. Introduction

The Yocto-Watt is a 58x50mm electronic module which allows you to measure by USB the power consumed by an electrical appliance. This tool is commonly called a wattmeter. It is also able to measure the voltage between two electric wires as well as the current going through a wire. It can also integrate the instantaneous consumption (in Watt) of an appliance to compute its hourly consumption (in Watt * hour).

The Yocto-Watt works with both direct current and alternating current. In the latter case, the Yocto-Watt is able to compute the voltage/current shift ($\cos \Phi$).

The module precision is of 0.002 W / 1.5% when measuring direct current and of 0.02 W / 1.5% when measuring alternating current. An important specificity of this module is to be an isolated wattmeter: the measuring part is electrically isolated from the USB part. This makes it possible to measure electric loads that do not share the same ground reference as the computer USB bus.

When mounted in a proper protective enclosure, the Yocto-Watt can even be used to measure the power consumption on appliances connected directly to the mains, because it features a safety isolation of 3kV r.m.s. between the USB interface and the measuring part. Caution, make sure to read carefully the safety recommendations below and the section describing the electric isolation before connecting the Yocto-Watt to a mains powered appliance.



The Yocto-Watt module

The Yocto-Watt 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-Watt 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-Watt is easy to install anywhere and easy to drive from a maximum of programming languages. If you are

nevertheless disappointed with this module, or if you need additional information, do not hesitate to contact Yoctopuce support:

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

1.1. Safety Information

The Yocto-Watt 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

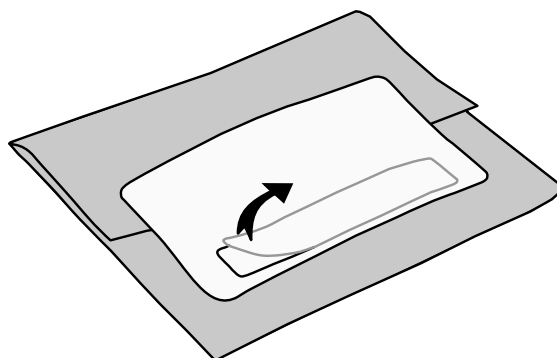
The Yocto-Watt should not be used without a protective enclosure, because of the accessible bare electronic components. For optimal safety, it should be put into a non-metallic, non-inflammable enclosure, resistant to a mechanical stress level of 5 J. For instance, use a polycarbonate (e.g. LEXAN) enclosure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

Maintenance

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

Identification

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



Identification label is integrated in the package label.

Application

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

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

Environment

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

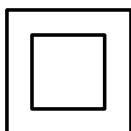
Hot surface and fire risk



When strong currents flow through the device, the PCB and some components may become hot, exceeding 60°C. A direct contact with the skin would result in bodily injury, and an inappropriate enclosure could create a serious fire hazard. When continuous exposure to strong currents is expected, pay attention to burns and fire hazard prevention. A fan is a good way to help with cooling, but in order to ensure that the system remains safe even if the fan is inoperative, it is essential to use an enclosure rated V-1 or V-0 according to IEC 60695-11-10 standard. Make sure that the enclosure openings are facing sideways or top, not bottom.

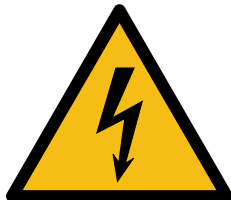
If you intend to use strong currents, make sure to use a power cable rated for high temperature, because the heat will propagate through the terminal blocks to the power cable.

IEC 61140 Protection Class II



This equipment is protected throughout by a double insulation or a reinforced insulation. The electronics on the side of USB connector are designed for safety extra-low voltages, while the electronics on the opposite side may carry hazardous voltages.

Electrical shock



Voltages above 60 V DC, 30 V AC r.m.s. or 42.4 V peak create electrocution and fire hazards. Since the Yocto-Watt has bare contacts, you should not connect it to hazardous voltage without using a proper enclosure, electrically insulated, non-flammable and shock-proof as described previously. A few extra precautions are required, that we have summarized in the paragraph below. Make sure to include these advices in the most adequate wording in the installation and maintenance manuals of your final product.

1.2. Electrocution Prevention Advice

The Yocto-Watt can be used without major risk with voltages up to 250 V AC r.m.s., as long as you adhere to a few safety guidelines and apply common sense. In order to help you to avoid an embarrassing nomination to the [Darwin Awards](#) or a charge of negligent homicide, we have summarized a few recommendations below:

- To attach the device, use **non-conductive plastic screw and spacers**. Fixing holes are under the safety clearance distance, so metal screw would be dangerous.
- Put the Yocto-Watt in a proper **insulated enclosure** *before* applying hazardous voltage. Doing so afterwards is no more possible. Conversely, make sure all hazardous wires are not live *before* opening the enclosure.
- Ensure that no object or liquid may accidentally enter into the enclosure.
- Ensure that all wires hold firmly in the terminal block, and that no thread escapes from the terminal block in case you use threaded cable.
- Check that all circuit connections are in accordance to the isolation voltage levels described in the Yocto-Watt description (see next chapter). USB and Ethernet wires should be well separated from any wire with a hazardous voltage.
- Do not work if you are tired. Brilliant decisions taken at 2 a.m. tend to lose their shine at first light.
- Do not work alone. If things go bad, you can then hope to be rescued fast enough.
- Use your head, your luck will not last forever.

- Hazardous voltage wiring is subject to local electrical standards. Some tasks may only be performed by qualified personnel.

1.3. Environmental conditions

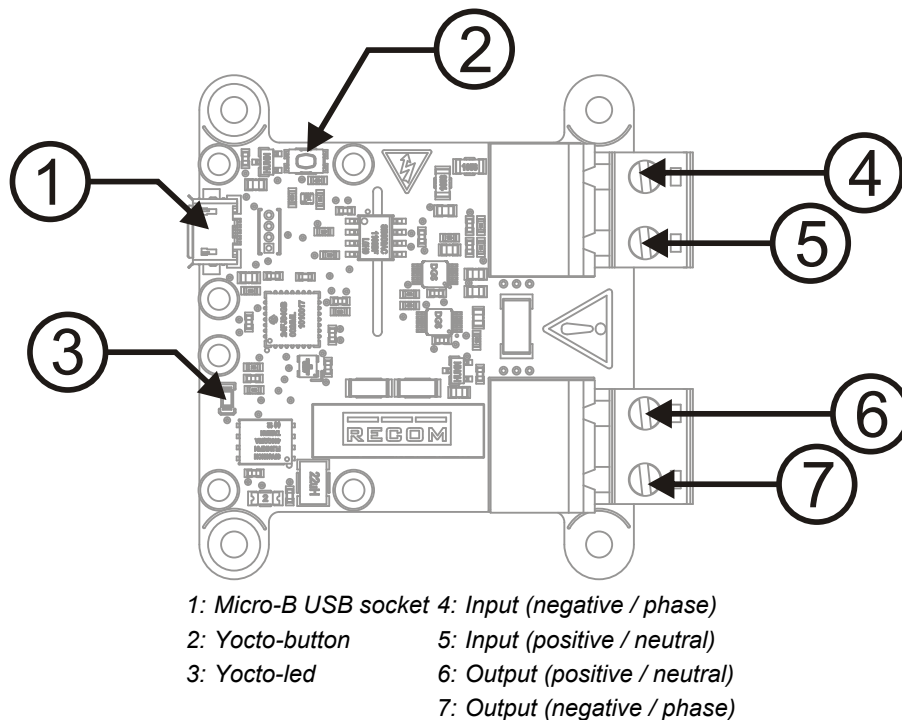
The Yocto-Watt is designed to avoid electrocution hazards caused by temporary overvoltage expected on wall outlets (**Overvoltage Category II**). It has not been designed for use in fixed installation and for industrial use with permanent connection to the fixed distribution circuit.

The safety insulation of the Yocto-Watt has 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 80% RH, non condensing**. Use in an environment with significant solid pollution or conductive pollution is not safe, unless the device is protected from such pollution using an IP67 or IP68 enclosure, which should only be opened in an unpolluted environment. The Yocto-Watt safety insulation was designed for use **up to altitude 2000m**. Usage at higher altitude require a thorough analysis of safety impacts based on the application.

The Yocto-Watt can be used on the whole specified voltage and current range without risk under normal temperature conditions according to IEC61010-1, i.e. 5°C to 40°C. The Yocto-Watt is also designed to be usable on an extended temperature range, but only when specific conditions are met to ensure safety: when the temperature sinks, there must be no condensing humidity on the device; when the temperature rises, in case the device is used with strong voltages, the ambient temperature around heat dissipating elements must stay stable to prevent thermal runaway.

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

2. Presentation



2.1. Common elements

All Yocto-modules share a number of common functionalities.

USB connector

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

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

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

A higher voltage is likely to destroy the device. The behaviour with a lower voltage is not specified, but it can result in 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-Watt modules, this number starts with YWATTMK1. 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.

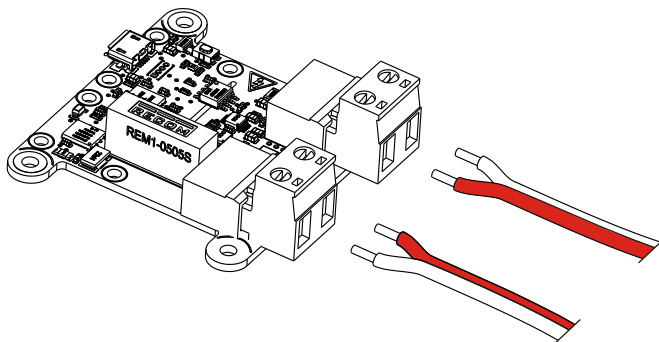
¹ short-short-short long-long-long short-short-short

² support@yoctopuce.com

2.2. Specific elements

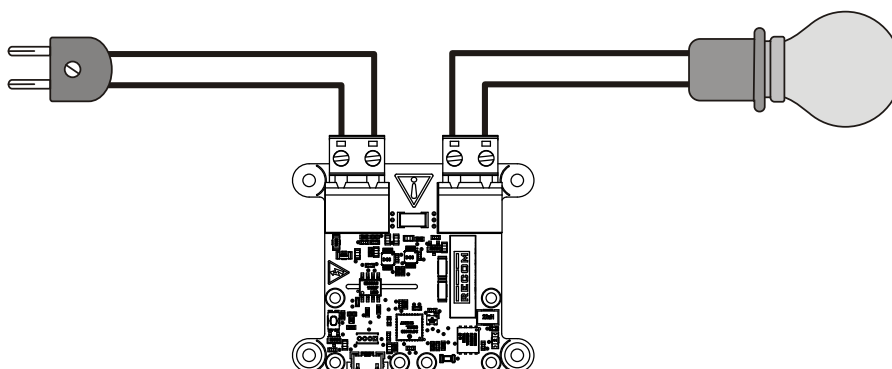
The terminal blocks

The terminal blocks of the Yocto-Watt can be separated into two parts, allowing you to easily fix plugs on the electric cables to then connect them in the Yocto-Watt.



The terminals can be split into two parts. Note the symmetry in the polarity.

The Yocto-Watt must be located between the power source and the power consumer, current is measured between the two poles at the center of the device. Note that the two sockets are symmetrical: if you get the polarity wrong, the Yocto-Watt will still work, but with direct current you will obtain negative voltage and current values. In the same way, if you connect the source on the output connector and the load on the input connector, with direct current you will obtain a negative current value.



The Yocto-Watt must be located between the source and the consumer.

Before making any connection, make sure to fully read the safety advices at the beginning of the manual, and the chapter named *Assembly and connections*.

The terminal block can hold solid or stranded wires between 24 AWG and 12 AWG, meaning 0.2 mm² to 3.31 mm². The wire must be stripped on 6mm, and the recommended tightening torque is 0.56 Nm.

If you intend to use strong currents or hazardous live voltages, make sure to use a power cable wire with a flammability RATING of UL 2556 VW-1 or equivalent, to prevent any risk of flame propagation.

The safety isolation of the Yocto-Watt makes it possible to perform measurements directly on mains circuits of measurement category II up to 250 V AC r.m.s. for the product rated environmental specifications.

The sensor

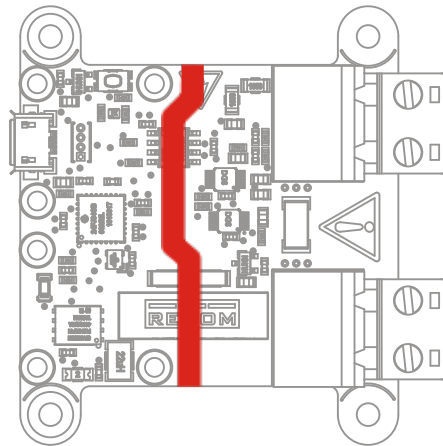
The Yocto-Watt is based on two analog/digital ADS1115IDGST converters produced by [Texas Instruments](#).

The input impedance for the voltage measurement is 331.2 KOhm. A voltage divider reduces the voltage to be measured.

To measure the current, the Yocto-Watt uses a metal shunt (a 5 mOhm resistor³), located between the two connectors. As the current increases, the temperature of the shunt rises as well. The shunt is designed to be self-cooled by convective heat transfer. At 10 Amps, the PCB temperature near the shunt is typically 15°C above ambient temperature. At 16 Amps, the PCB temperature near the shunt is typically 35°C above ambient temperature⁴. This is the expected behaviour for the shunt to dissipate power. However, for the convective heat transfer to work properly with high amperage, you should make sure that the Yocto-Watt is kept in a position such that air can easily flow. If you use the device in an enclosure, make sure the enclosure is properly vented, to ensure proper operation of the device and prevent fire hazards. For a continuous use À partir de 10 Amps, the use of a small fan is recommended. But don't forget that in any case your system should be designed to prevent any fire risk even in case of failure of the fan (single abnormal condition). Use an enclosure rated V-1 or V-0 according to IEC 60695-11-10 standard, and make sure that the enclosure openings are facing sideways or top, not bottom.

2.3. Safety isolation

The Yocto-Watt is designed as two distinct electrical circuits, separated by a reinforced isolation⁵. This isolation plays an essential role for the safety of system users.



Yocto-Watt safety isolation

On the one side of the isolation where the USB connector can be found, components connectors and cables connected to the system are accessible to the user without risk: the circuit power supply is floating, protected by a double isolation or a reinforced isolation, and voltage levels are below 6V (*Safety Extra Low Voltage*).

On the side with the terminal blocks, connected cables may carry hazardous live voltage, depending on the application. So unless you are only connecting to non-hazardous voltage, make sure that **no conductive object is brought nearer than 3mm from the hazardous side of the device**, even accidentally. Pay attention to the following items:

- To fix the Yocto-Watt on the hazardous side, **use isolating plastic screw and spacers only**. You will find PPS / PEEK screws and spacers in the Yoctopuce shop, that are resistant to high temperatures.
- If you intend to use a metal enclosure, make sure that it does not come nearer than 3mm from the hazardous side of the Yocto-Watt, and that it prevents any access to the hazardous side.
- In order to properly protect from hazardous voltages, a solid isolation should be at least 0.4mm thick.

³ 5 mOhm shunt starting from Revision B. Earlier revisions used a 10 mOhm shunt.

⁴ these temperatures apply to revision B and later of the product. The shunt temperature on earlier revisions may rise even higher.

⁵ This description of the isolation applies to \$SPECREV\$ of the product. Some earlier revisions of the product had a weaker isolation.

The specifications of the safety isolation are as follows:

- **Test voltage:** 3kV r.m.s.
- **Clearance distance:** 3.8mm
- **Creepage distance:** 3.8 / 6mm
- **Material group:** Cat. I / IIIa

The specifications of the isolation between terminal block contacts are as follow:

- **Dielectric strength:** 1.5kV
- **Clearance distance:** 3mm
- **Creepage distance:** 3mm
- **Material group:** Cat IIIa

2.4. Optional accessories

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

Screws and spacers

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

Micro-USB hub

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

YoctoHub-Ethernet, YoctoHub-Wireless and YoctoHub-GSM

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

1.27mm (or 1.25mm) connectors

In case you wish to connect your Yocto-Watt 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.

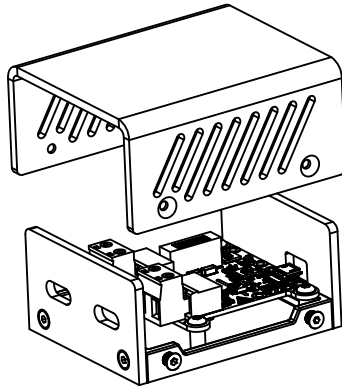
You can mount the Yocto-Watt directly on the hub using screw and spacers, and connect it using 1.27mm board-to-board connectors. To prevent shortcuts, it is best to solder the female connector on the hub and the male connector on the Yocto-Watt.

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-Watt has been designed to be installed as is in your project. Nevertheless, Yoctopuce sells enclosures specifically designed for Yoctopuce devices. More details are available on the

Yoctopuce web site ⁶. The suggested enclosure model for your Yocto-Watt is the YoctoBox-Watt-Transp.



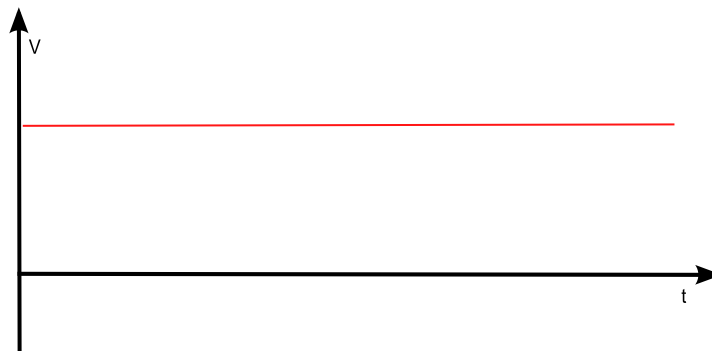
You can install your Yocto-Watt in an optional enclosure

⁶ <http://www.yoctopuce.com/EN/products/category/enclosures>

3. Direct current vs. alternating current

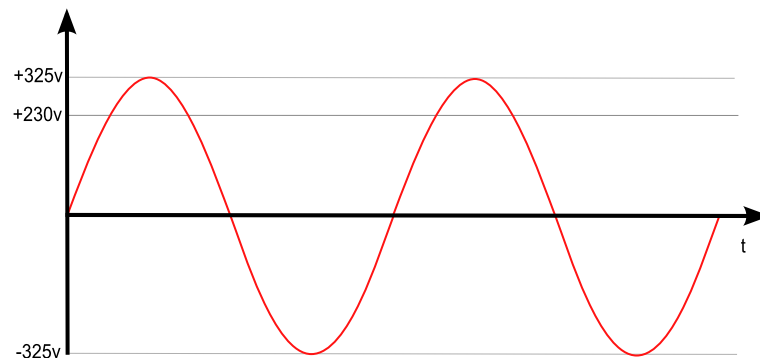
3.1. Measure of the voltage

The Yocto-Watt module is able to measure a direct voltage as well as an alternating voltage. You can typically measure a direct voltage on the terminals of a battery or at the output of a stabilized power supply for electronic equipment. It is supposed to stay relatively stable over time.



Direct voltage, stable over time

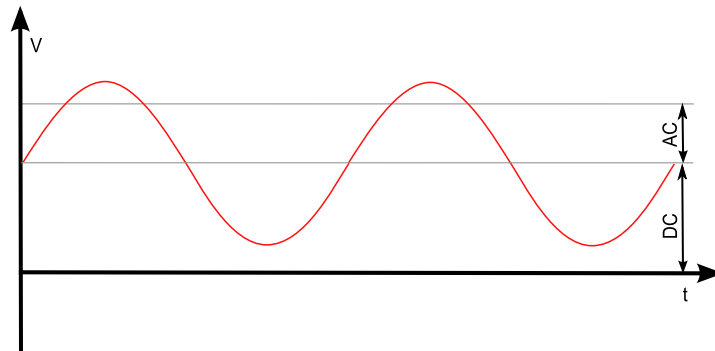
An alternating voltage is a voltage which periodically varies over time. The most typical example is the voltage of the mains. In Europe, the voltage of the mains varies between -325V and +325V about 50 times per second, for an "typical value"¹ of 230V. A dynamo such as the one on your bike also produces alternating current.



Alternating voltage, for example the voltage of the mains in Europe: it varies periodically.

¹ The exact term is "RMS value"

Note that an alternating voltage is not necessarily symmetrical. One can then describe it as the sum of a direct voltage and of an alternating symmetrical voltage.



Asymmetrical alternating voltage. It is the sum of a direct voltage and of an alternating voltage.

For this reason, the Yocto-Watt provides voltage measures in the shape of a measure of the direct voltage (DC) coupled with a measure of the alternating voltage (AC). When you measure a voltage with a Yocto-Watt, several distinct cases can occur:

- A non null DC value and a null AC value: you are measuring a direct voltage.
- A non null AC value and an almost null DC value: you are measuring a symmetrical alternating voltage.
- A non null AC value and a non null DC value: you are measuring an alternating voltage which is not symmetrical.
- A non null stable DC value and a completely random AC value: you are measuring a voltage which is not stabilized, typically a (somewhat too) cheap AC adapter.

The Yocto-Watt has a 0.002 W / 1.5% accuracy for direct voltage. The accuracy is especially good under 50V DC. However, its accuracy for alternating voltage is not as good: take into account an error of more or less 0.02 W / 1.5% at 50-60Hz. Below 30Hz, the lower the frequency is, the less stable the measure is. Above 80Hz, the higher the frequency is, the more underestimated the measure is. Note that the voltage AC measure given by the Yocto-Watt corresponds to the [RMS value](#) (and not to the peak amplitude).

3.2. Measure of the current

The Yocto-Watt measure of the current is performed in a way similar to the voltage measure. When the voltage measured is direct, the Yocto-Watt does a DC current measure only. With an alternating voltage (AC), the Yocto-Watt performs an AC current measure with a DC component (usually null) and an AC component, given as an [RMS value](#).

3.3. Measure of the power

Power is usually computed as the product of the voltage at the terminals of an appliance and the current going through it. With direct current, it is enough to multiply the DC voltage measure by the DC current measure to compute the consumed power.

With alternating current, it is not as simple. Multiplying the AC voltage by the AC current enables you to compute the apparent power of the appliance (in VA), but not the power actually consumed. Indeed, to compute the consumed power, it is absolutely necessary to take into account that current and voltage do not necessarily oscillate at the same time. This is what happens with *inductive* loads (see below). The correct method to measure the power effectively consumed consists in integrating the product *current * voltage*, on the basis of samples measured several hundreds of times per second. This is what the Yocto-Watt does.

Inductive load

An inductive load, in the opposite to a purely resistive load, is a load which has the specificity of inducing a shift between voltage and current. Schematically, an inductive load restitutes part of the

energy it has absorbed. This is why the consumed energy cannot be computed as the product of the current/voltage averages at the terminals of this load.

Appliances representing inductive loads are often characterized by the presence of windings (transformers, motors, ...). Most of the appliances working on the mains are inductive loads.

Power factor

The ration between the power effectively consumed and the apparent power is called the power factor, or $\cos \varphi$. A 1.0 power factor indicated a purely resistive load. The more the power factor approaches zero, the more inductive is the load.

A filament bulb has a 1.0 power factor. An electrical transformer (power supply) typically has a 0.7 power factor, which means that it consumes only 70% of the energy it is provided, the remainder being restituted to the electrical network.

The Yocto-Watt provides an estimate of the power factor in real time.

4. 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.

4.1. Prerequisites

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

A computer

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

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

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

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

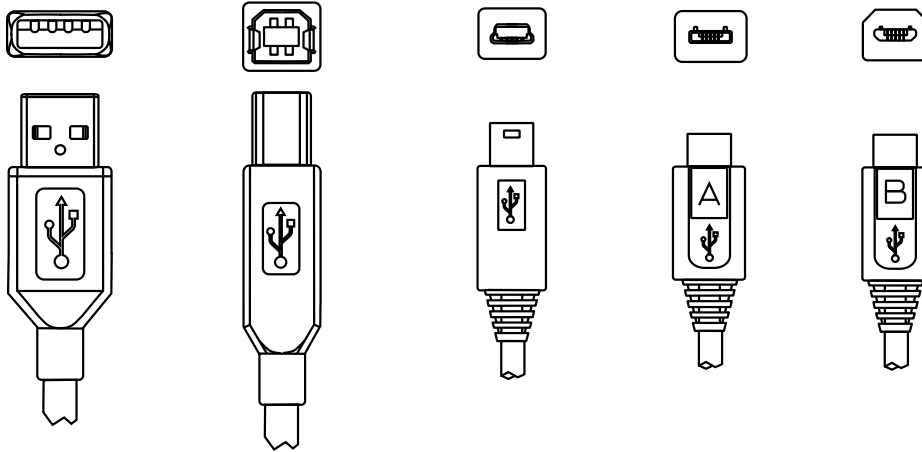
Linux kernels currently supported are the 2.6 branch, the 3.x branch and the 4.x branch. Other versions of the Linux kernel, and even other UNIX variants, are very likely to work as well, as Linux support is implemented through the standard **libusb** API. Yoctopuce is frequently testing its modules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

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

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

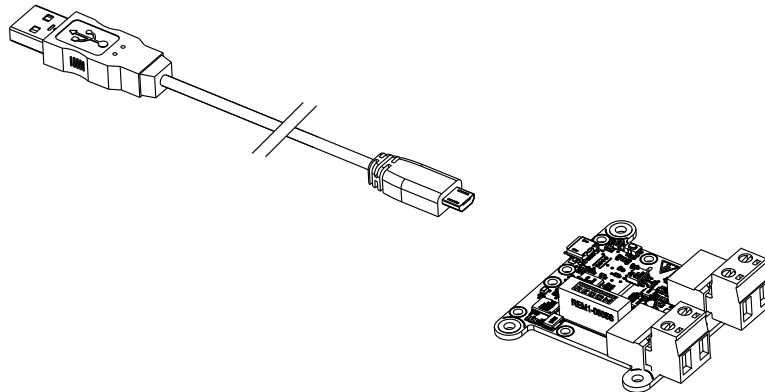
A USB 2.0 cable, type A-micro B

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



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

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



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

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

4.2. Testing USB connectivity

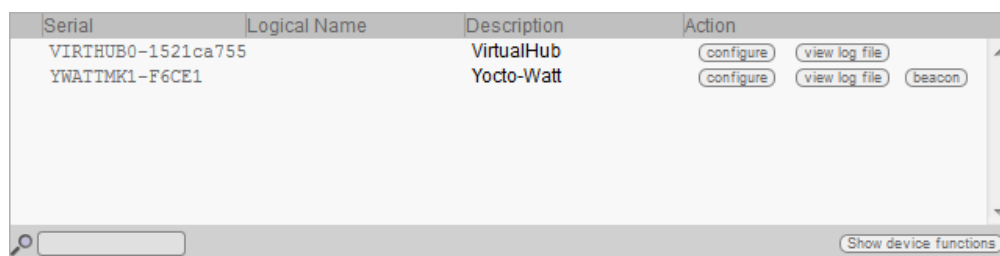
At this point, your Yocto-Watt should be connected to your computer, which should have recognized it. It is time to make it work.

Go to the Yoctopuce web site and download the *Virtual Hub* software³. It is available for Windows, Linux, and Mac OS X. Normally, the Virtual Hub software serves as an abstraction layer for languages which cannot access the hardware layers of your computer. However, it also offers a succinct interface to configure your modules and to test their basic functions. You access this

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

³ www.yoctopuce.com/EN/virtualhub.php

interface with a simple web browser⁴. Start the *Virtual Hub* software in a command line, open your preferred web browser and enter the URL `http://127.0.0.1:4444`. The list of the Yoctopuce modules connected to your computer is displayed.



Serial	Logical Name	Description	Action
VIRTHUB0-1521ca755		VirtualHub	configure view log file
YWATTMK1-F6CE1		Yocto-Watt	configure view log file beacon

At the bottom of the interface, there is a search bar and a button labeled "Show device functions".

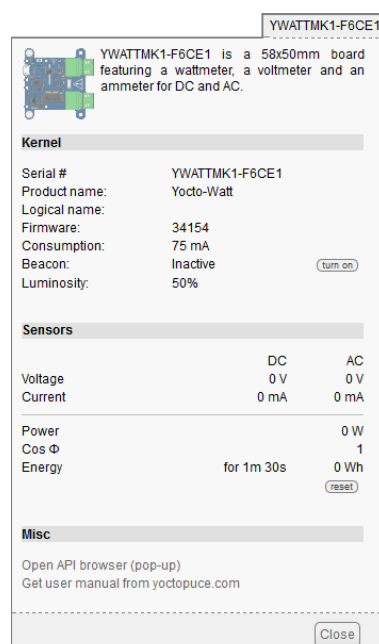
Module list as displayed in your web browser

4.3. 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.

4.4. 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-Watt.



YWATTMK1-F6CE1

YWATTMK1-F6CE1 is a 58x50mm board featuring a wattmeter, a voltmeter and an ammeter for DC and AC.

Kernel

Serial # YWATTMK1-F6CE1
 Product name: Yocto-Watt
 Logical name:
 Firmware: 34154
 Consumption: 75 mA
 Beacon: Inactive [turn on](#)
 Luminosity: 50%

Sensors

	DC	AC
Voltage	0 V	0 V
Current	0 mA	0 mA
Power		0 W
Cos Φ		1
Energy	for 1m 30s 0 Wh reset	

Misc

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

[Close](#)

Properties of the Yocto-Watt module

This windows allows you, among other things, to play with your module to check that it is working correctly, as measured values are displayed in real time.

4.5. Configuration

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

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

YWATTK1-F6CE1

Edit parameters for device YWATTK1-F6CE1, and click on the Save button.

Serial # YWATTK1-F6CE1

Product name Yocto-Watt

Firmware 34154

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.

YWATTK1-F6CE1.current1 /

YWATTK1-F6CE1.current2 /

YWATTK1-F6CE1.power /

YWATTK1-F6CE1.voltage1 /

YWATTK1-F6CE1.voltage2 /

YWATTK1-F6CE1.dataLogger /

Datalogger and Timed reports

Timed reports enabled on all functions
Recording is disabled
no recorded data

Yocto-Watt module configuration.

Firmware

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

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

Logical name of the module

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

Luminosity

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

Logical names of functions

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

The five functions provided by the Yocto-Watt module are the *power*, *current1*, *current2*, *voltage1*, and *voltage2* functions. *current1* and *current2* correspond to the functions measuring DC and AC current respectively. *voltage1* and *voltage2* correspond to the voltage measures for DC and AC respectively. Simply click on the corresponding "rename" button to assign a new logical name to one of the functions.

⁵ www.yoctopuce.com/EN/virtualhub.php

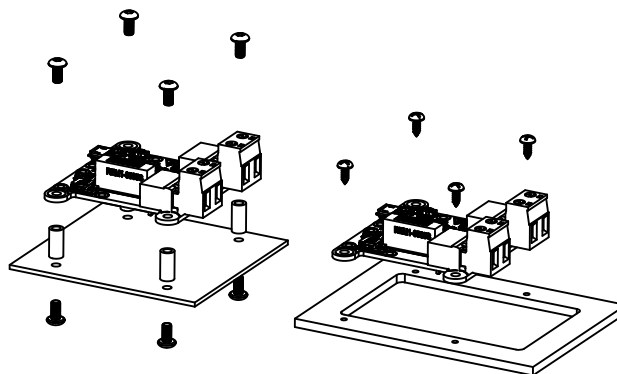
⁶ More information available in the virtual hub documentation

5. Assembly and connections

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

5.1. Fixing

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



Examples of assembly on supports

The Yocto-Watt module contains 3mm assembly holes. You can use these holes for screws. The screw head diameter must not be larger than 8mm 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.

5.2. 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.

5.3. Electromagnetic compatibility (EMI)

Connection methods to integrate the Yocto-Watt 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

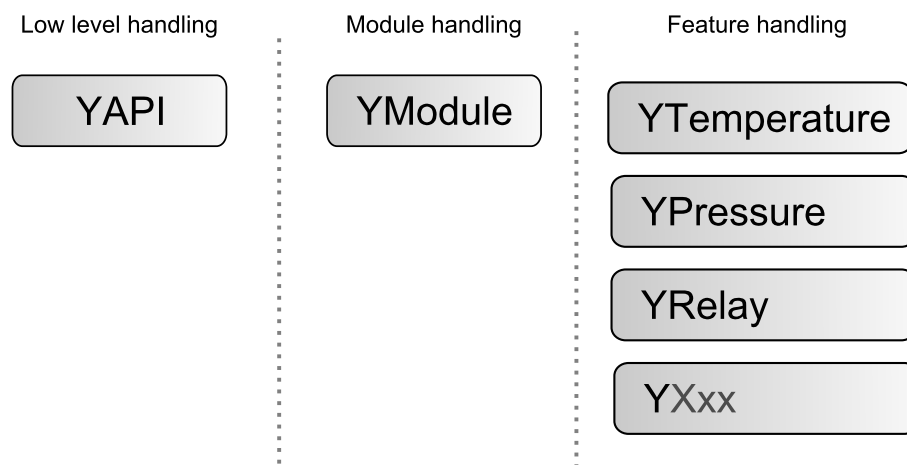
6. Programming, general concepts

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

6.1. Programming paradigm

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

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



Structure of the Yoctopuce API.

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-Watt is viewed as a module hosting a given number of functions. In the API, these functions are objects which can be found independently, in several ways.

Access to the functions of a module

Access by logical name

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

Access by enumeration

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

Access by hardware name

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

Difference between *Find* and *First*

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

Function handling

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

Access to the modules

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

Functions/Module interaction

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

6.2. The Yocto-Watt module

The Yocto-Watt module provides a two instances of the Voltage function. The `voltage1` input corresponds to the DC voltage component, while the `voltage2` input corresponds to the AC voltage component. The typical accuracy of the DC measure is 0.01V or 0.5 percent, while the precision of the AC measure is 0.1V or 3 percent, for frequencies between 30Hz and 80Hz.

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

power : Power

attribute	type	modifiable ?
logicalName	String	modifiable
advertisedValue	String	modifiable
unit	String	read-only
currentValue	Fixed-point number	read-only
lowestValue	Fixed-point number	modifiable
highestValue	Fixed-point number	modifiable
currentRawValue	Fixed-point number	read-only
logFrequency	Frequency	modifiable
reportFrequency	Frequency	modifiable
advMode	Enumerated	modifiable
calibrationParam	Calibration parameters	modifiable
resolution	Fixed-point number	modifiable
sensorState	Integer	read-only
cosPhi	Fixed-point number	read-only
meter	Fixed-point number	modifiable
deliveredEnergyMeter	Fixed-point number	read-only
receivedEnergyMeter	Fixed-point number	read-only
meterTimer	Integer	read-only

voltage1 : Voltage

voltage2 : Voltage

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

lowestValue	Fixed-point number	modifiable
highestValue	Fixed-point number	modifiable
currentRawValue	Fixed-point number	read-only
logFrequency	Frequency	modifiable
reportFrequency	Frequency	modifiable
advMode	Enumerated	modifiable
calibrationParam	Calibration parameters	modifiable
resolution	Fixed-point number	modifiable
sensorState	Integer	read-only
enabled	Boolean	modifiable

current1 : Current**current2 : Current**

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

6.3. 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-Watt module, this serial number always starts with YWATTMK1. 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 42 at the factory.

productRelease

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

firmwareRelease

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

persistentSettings

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

luminosity

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

beacon

Activity of the localization beacon of the module.

upTime

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

usbCurrent

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

rebootCountdown

Countdown to use for triggering a reboot of the module.

userVar

32bit integer variable available for user storage.

6.4. Power

electrical power sensor control interface, available for instance in the Yocto-Watt

The `YPower` class allows you to read and configure Yoctopuce electrical power sensors. It inherits from `YSensor` class the core functions to read measurements, to register callback functions, and to access the autonomous datalogger. This class adds the ability to access the energy counter and the power factor.

logicalName

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

unit

Short character string representing the measuring unit for the electrical power.

currentValue

Current value of the electrical power, in Watt, as a floating point number.

lowestValue

Minimal value of the electrical power, in Watt, as a floating point number.

highestValue

Maximal value of the electrical power, in Watt, as a floating point number.

currentRawValue

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

logFrequency

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

reportFrequency

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

advMode

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

calibrationParam

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

resolution

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

sensorState

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

cosPhi

Power factor, i.e. ratio between the real power consumed (in W) and the apparent power provided (VA).

meter

Energy counter, maintained by the wattmeter by integrating the power consumption over time. Note that this counter is reset at each start of the device.

deliveredEnergyMeter

Delivered energy counter, maintained by the wattmeter by integrating the power consumption over time when positive only. Note that this counter is reset at each start of the device.

receivedEnergyMeter

Received energy counter, maintained by the wattmeter by integrating the power consumption over time when negative only. Note that this counter is reset at each start of the device.

meterTimer

Elapsed time since last energy counter reset, in seconds.

6.5. Voltage

voltage sensor control interface, available for instance in the Yocto-Motor-DC, the Yocto-Volt or the Yocto-Watt

The `YVoltage` class allows you to read and configure Yoctopuce voltage sensors. It inherits from `YSensor` class the core functions to read measurements, to register callback functions, and to access the autonomous datalogger.

logicalName

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

unit

Short character string representing the measuring unit for the voltage.

currentValue

Current value of the voltage, in Volt, as a floating point number.

lowestValue

Minimal value of the voltage, in Volt, as a floating point number.

highestValue

Maximal value of the voltage, in Volt, as a floating point number.

currentRawValue

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

logFrequency

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

reportFrequency

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

advMode

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

calibrationParam

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

resolution

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

sensorState

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

enabled

Activation state of the input.

6.6. Current

current sensor control interface, available for instance in the Yocto-Amp, the Yocto-Motor-DC or the Yocto-Watt

The `YCurrent` class allows you to read and configure Yoctopuce current sensors. It inherits from `YSensor` class the core functions to read measurements, to register callback functions, and to access the autonomous datalogger.

logicalName

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

unit

Short character string representing the measuring unit for the current.

currentValue

Current value of the current, in mA, as a floating point number.

lowestValue

Minimal value of the current, in mA, as a floating point number.

highestValue

Maximal value of the current, in mA, as a floating point number.

currentRawValue

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

logFrequency

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

reportFrequency

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

advMode

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

calibrationParam

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

resolution

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

sensorState

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

enabled

Activation state of the input.

6.7. 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 slower than usual.

usage

Percentage of datalogger memory in use.

clearHistory

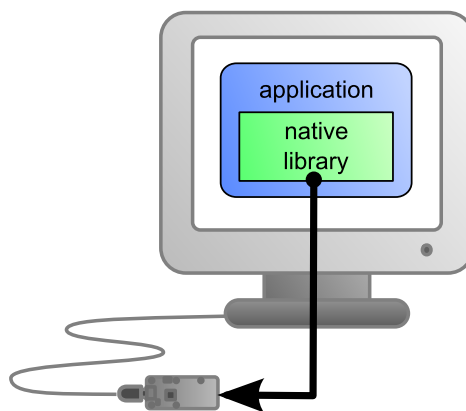
Attribute that can be set to true to clear recorded data.

6.8. What interface: Native, DLL or Service ?

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

Native control

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

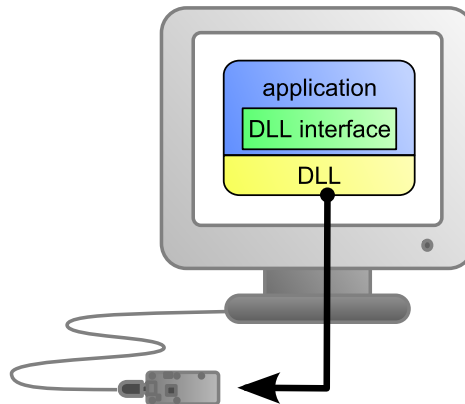


The application uses the native library to control the locally connected module

Native control by DLL

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

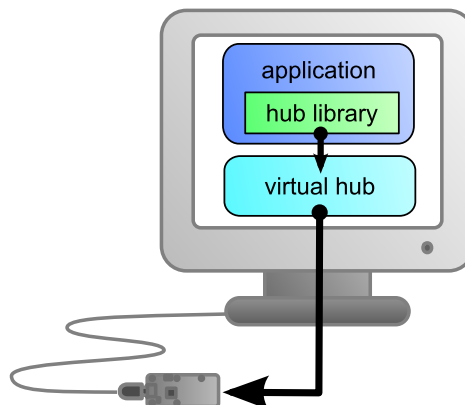
in a given language. Indeed, the "useful" part of the control code is located in the DLL which is the same for all languages: the effort to support a new language is limited to coding the small library which controls the DLL. From the end user stand point, there are few differences: one must simply make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



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

Control by service

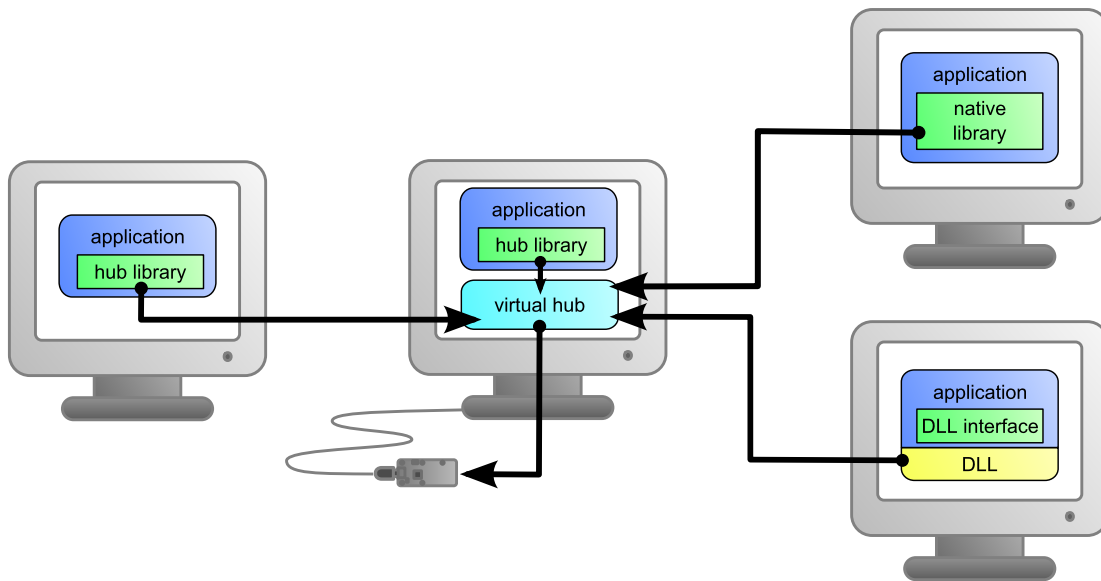
Some languages do simply not allow you to easily gain access to the hardware layers of the machine. It is the case for Javascript, for instance. To deal with this case, Yoctopuce provides a solution in the form of a small piece of software called *VirtualHub*¹. It can access the modules, and your application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



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

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

¹ www.yoctopuce.com/EN/virtualhub.php



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

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

Language	Native	Native with DLL	VirtualHub
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

Limitations of the Yoctopuce libraries

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

6.9. Programming, where to start?

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

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

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-Watt.

7. Using the Yocto-Watt in command line

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

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

7.1. Installing

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

```
C:\>YPower any get_currentValue
```

To use the command API on Linux, you need either have root privileges or to define an *udev* rule for your system. See the *Troubleshooting* chapter for more details.

7.2. Use: general description

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

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

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

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

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

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

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

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

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

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

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

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

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

7.3. Control of the Power function

To control the Power function of your Yocto-Watt, you need the YPower executable file.

For instance, you can launch:

```
C:\>YPower any get_currentValue
```

This example uses the "*any*" target to indicate that we want to work on the first Power 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-Watt module with the *YWATTMK1-123456* serial number which you have called "*MyModule*", and its power 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:\>YPower YWATTMK1-123456.power describe
C:\>YPower YWATTMK1-123456.MyFunction describe
C:\>YPower MyModule.power describe
C:\>YPower MyModule.MyFunction describe
C:\>YPower MyFunction describe
```

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

```
C:\>YPower all describe
```

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

```
C:\>YPower /help
```

7.4. Control of the module part

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

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

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

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

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

```
C:\>YModule YWATTMK1-12346 set_logicalName MonPremierModule
C:\>YModule YWATTMK1-12346 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 YWATTMK1-12346 set_logicalName MonPremierModule
C:\>YModule YWATTMK1-12346 saveToFlash
```

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

```
C:\>YModule -s YWATTMK1-12346 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.

7.5. Limitations

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

You can easily work around this limitation by using a Virtual Hub: run the VirtualHub³ on the concerned machine, and use the executables of the command line API with the `-r` option. For example, if you use:

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

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

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

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

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

8. Using the Yocto-Watt with Python

Python is an interpreted object oriented language developed by Guido van Rossum. Among its advantages is the fact that it is free, and the fact that it is available for most platforms, Windows as well as UNIX. It is an ideal language to write small scripts on a napkin. The Yoctopuce library is compatible with Python 2.6+ and 3+. It works under Windows, Mac OS X, and Linux, Intel as well as ARM. The library was tested with Python 2.6 and Python 3.2. Python interpreters are available on the Python web site¹.

8.1. 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.

8.2. Dynamic library

A section of the low-level library is written in C, but you should not need to interact directly with it: it is provided as a DLL under Windows, as a .so files under UNIX, and as a *.dylib* file under Mac OS X. Everything was done to ensure the simplest possible interaction from Python: the distinct versions of the dynamic library corresponding to the distinct operating systems and architectures are stored in the *cdll* directory. The API automatically loads the correct file during its initialization. You should not have to worry about it.

If you ever need to recompile the dynamic library, its complete source code is located in the Yoctopuce C++ library.

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

8.3. Control of the Power function

A few lines of code are enough to use a Yocto-Watt. Here is the skeleton of a Python code snippet to use the Power 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
power = YPower.FindPower("YWATMK1-123456.power")

# Hot-plug is easy: just check that the device is online
if power.isOnline():
    # Use power.get_currentValue()
    [...]

[...]
```

Let's look at these lines in more details.

YAPI.RegisterHub

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

YPower.FindPower

The `YPower.FindPower` function allows you to find a electrical power sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-Watt module with serial number `YWATMK1-123456` which you have named "*MyModule*", and for which you have given the *power* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
power = YPower.FindPower("YWATMK1-123456.power")
power = YPower.FindPower("YWATMK1-123456.MyFunction")
power = YPower.FindPower("MyModule.power")
power = YPower.FindPower("MyModule.MyFunction")
power = YPower.FindPower("MyFunction")
```

`YPower.FindPower` returns an object which you can then use at will to control the electrical power sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `YPower.FindPower` allows you to obtain the power consumption, as measured by the sensors. The return value is a number directly representing the value in Watts.

A real example

Launch Python and open the corresponding sample script provided in the directory **Examples/Doc-GettingStarted-Yocto-Watt** 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_power import *

def usage():
    scriptname = os.path.basename(sys.argv[0])
    print("Usage:")
    print(scriptname + ' <serial_number>')
    print(scriptname + ' <logical_name>')
    print(scriptname + ' any ')
    sys.exit()

def die(msg):
    sys.exit(msg + ' (check USB cable)')

errmsg = YRefParam()

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

target = sys.argv[1]

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

if target == 'any':
    # retrieve any Power sensor
    sensor = YPower.FirstPower()
    if sensor is None:
        die('No module connected')
else:
    sensor = YPower.FindPower(target + '.power')

if not (sensor.isOnline()):
    die('device not connected')
while sensor.isOnline():
    print("Power : " + "%2.1f" % sensor.get_currentValue() + "W (Ctrl-C to stop)")
    YAPI.Sleep(1000)
YAPI.FreeAPI()

```

8.4. 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()
```

8.5. Error handling

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

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

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

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

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

9. Using Yocto-Watt with C++

C++ is not the simplest language to master. However, if you take care to limit yourself to its essential functionalities, this language can very well be used for short programs quickly coded, and it has the advantage of being easily ported from one operating system to another. Under Windows, all the examples and the project models are tested with Microsoft Visual Studio 2010 Express, freely available on the Microsoft web site¹. Under Mac OS X, all the examples and project models are tested with XCode 4, available on the App Store. Moreover, under Mac OS X and under Linux, you can compile the examples using a command line with GCC using the provided `GNUmakefile`. In the same manner under Windows, a `Makefile` allows you to compile examples using a command line, fully knowing the compilation and linking arguments.

Yoctopuce C++ libraries² are integrally provided as source files. A section of the low-level library is written in pure C, but you should not need to interact directly with it: everything was done to ensure the simplest possible interaction from C++. The library is naturally also available as binary files, so that you can link it directly if you prefer.

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

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

9.1. Control of the Power function

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

```
#include "yocto_api.h"
#include "yocto_power.h"

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

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

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

² www.yoctopuce.com/EN/libraries.php

```
YPower *power;
power = YPower::FindPower("YWATTMK1-123456.power");

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

Let's look at these lines in more details.

yocto_api.h et yocto_power.h

These two include files provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api.h` must always be used, `yocto_power.h` is necessary to manage modules containing an electrical power sensor, such as Yocto-Watt.

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.

YPower::FindPower

The `YPower::FindPower` function allows you to find an electrical power sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-Watt module with serial number `YWATTMK1-123456` which you have named `"MyModule"`, and for which you have given the `power` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
YPower *power = YPower::FindPower("YWATTMK1-123456.power");
YPower *power = YPower::FindPower("YWATTMK1-123456.MyFunction");
YPower *power = YPower::FindPower("MyModule.power");
YPower *power = YPower::FindPower("MyModule.MyFunction");
YPower *power = YPower::FindPower("MyFunction");
```

`YPower::FindPower` returns an object which you can then use at will to control the electrical power sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `yFindPower` allows you to obtain the power consumption, as measured by the sensors. The return value is a number directly representing the value in Watts.

A real example

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

```

#include <stdlib.h>

using namespace std;

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

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

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

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

    if (target == "any") {
        psensor = YPower::FirstPower();
        if (psensor == NULL) {
            cout << "No module connected (check USB cable)" << endl;
            return 1;
        }
    } else {
        psensor = YPower::FindPower(target + ".Power");
    }

    while (1) {
        if (!psensor->isOnline()) {
            cout << "Module not connected (check identification and USB cable)";
            break;
        }
        cout << "Current Power: " << psensor->get_currentValue() << " W" << endl;
        cout << "      (press Ctrl-C to exit)" << endl;
        YAPI::Sleep(1000, errmsg);
    };
    YAPI::FreeAPI();

    return 0;
}

```

9.2. Control of the module part

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

```

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

#include "yocto_api.h"

using namespace std;

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

```

```

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

    // Setup the API to use local USB devices
    if(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_xxxx()` method. For more details regarding the used functions, refer to the API chapters.

Changing the module settings

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

```

#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;
}

```

9.3. Error handling

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

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

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

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

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

9.4. Integration variants for the C++ Yoctopuce library

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

Integration in source format (recommended)

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

- It guaranties the respect of the compilation conventions of your project (32/64 bits, inclusion of debugging symbols, unicode or ASCII characters, etc.);
- It facilitates debugging if you are looking for the cause of a problem linked to the Yoctopuce library;
- It reduces the dependencies on third party components, for example in the case where you would need to recompile this project for another architecture in many years;

- It does not require the installation of a dynamic library specific to Yoctopuce on the final system, everything is in the executable.

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

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

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

Integration as a static library

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

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

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

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

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

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

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

Integration as a dynamic library

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

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

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

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

- For Windows: **yocto.lib**

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

With GCC, the command line to compile is simply:

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

10. Using Yocto-Watt with C#

C# (pronounced C-Sharp) is an object-oriented programming language promoted by Microsoft, it is somewhat similar to Java. Like Visual-Basic and Delphi, it allows you to create Windows applications quite easily. All the examples and the project models are tested with Microsoft C# 2010 Express, freely available on the Microsoft web site¹.

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

10.1. Installation

Download the Visual C# Yoctopuce library from the Yoctopuce web site². There is no setup program, simply copy the content of the zip file into the directory of your choice. You mostly need the content of the `Sources` directory. The other directories contain the documentation and a few sample programs. All sample projects are Visual C# 2010, projects, if you are using a previous version, you may have to recreate the projects structure from scratch.

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

The Visual C#.NET Yoctopuce library is composed of a DLL and of source files in Visual C#. The DLL is not a .NET DLL, but a classic DLL, written in C, which manages the low level communications with the modules³. The source files in Visual C# manage the high level part of the API. Therefore, you need both this DLL and the .cs files of the `sources` directory to create a project managing Yoctopuce modules.

Configuring a Visual C# project

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

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

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

² www.yoctopuce.com/EN/libraries.php

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

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

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

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

10.3. Control of the Power function

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

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

// Retrieve the object used to interact with the device
YPower power = YPower.FindPower("YWATMK1-123456.power");

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

Let's look at these lines in more details.

YAPI.RegisterHub

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

YPower.FindPower

The `YPower.FindPower` function allows you to find a electrical power sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-Watt module with serial number `YWATMK1-123456` which you have named `"MyModule"`, and for which you have given the `power` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
power = YPower.FindPower("YWATMK1-123456.power");
power = YPower.FindPower("YWATMK1-123456.MyFunction");
power = YPower.FindPower("MyModule.power");
power = YPower.FindPower("MyModule.MyFunction");
power = YPower.FindPower("MyFunction");
```

`YPower.FindPower` returns an object which you can then use at will to control the electrical power sensor.

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

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `YPower.FindPower` allows you to obtain the power consumption, as measured by the sensors. The return value is a number directly representing the value in Watts.

A real example

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

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

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

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

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

            YPower psensor;

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

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

            if (target == "ANY") {
                psensor = YPower.FirstPower();

                if (psensor == null) {
                    Console.WriteLine("No module connected (check USB cable) ");
                    Environment.Exit(0);
                }
            } else {
                psensor = YPower.FindPower(target + ".power");
            }

            if (!psensor.isOnline()) {
                Console.WriteLine("Module not connected");
                Console.WriteLine("check identification and USB cable");
                Environment.Exit(0);
            }

            while (psensor.isOnline()) {
                Console.WriteLine("Current power: " + psensor.get_currentValue().ToString())
            }
        }
    }
}
```

```

        + " W");
        Console.WriteLine("  (press Ctrl-C to exit)");

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

```

10.4. 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.Write("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();
        }
    }
}
```

10.5. Error handling

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

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

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

- If your code catches the exception and handles it, everything goes well.

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

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

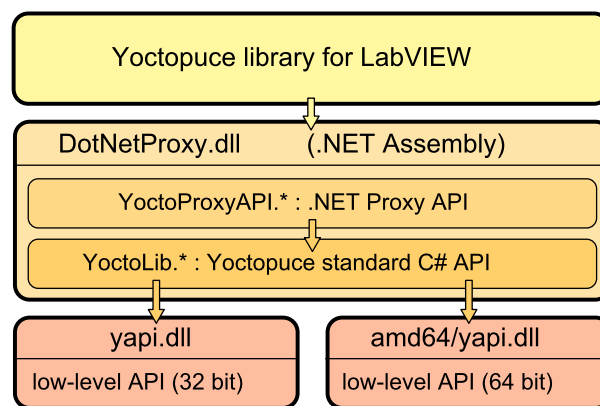
When you work without exceptions, you can obtain an error code and an error message explaining the source of the error. You can request them from the object which returned the error, calling the `errType()` and `errMessage()` methods. Their returned values contain the same information as in the exceptions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

11. Using the Yocto-Watt 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.

11.1. 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.

11.2. 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.

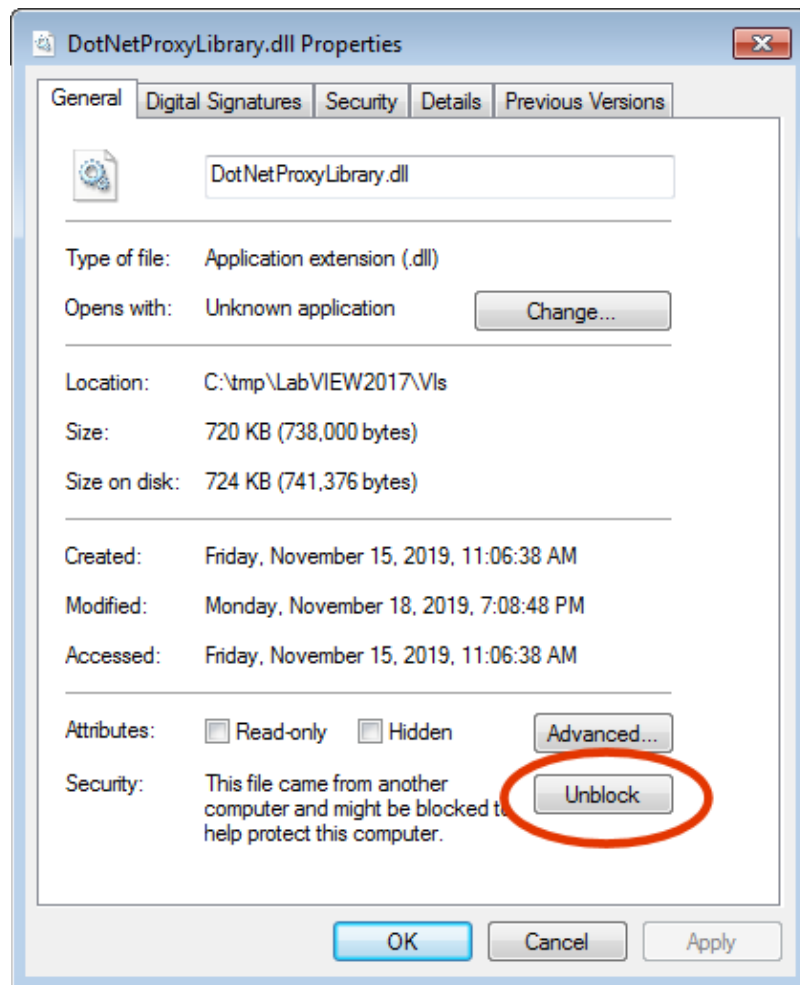
11.3. Installation

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

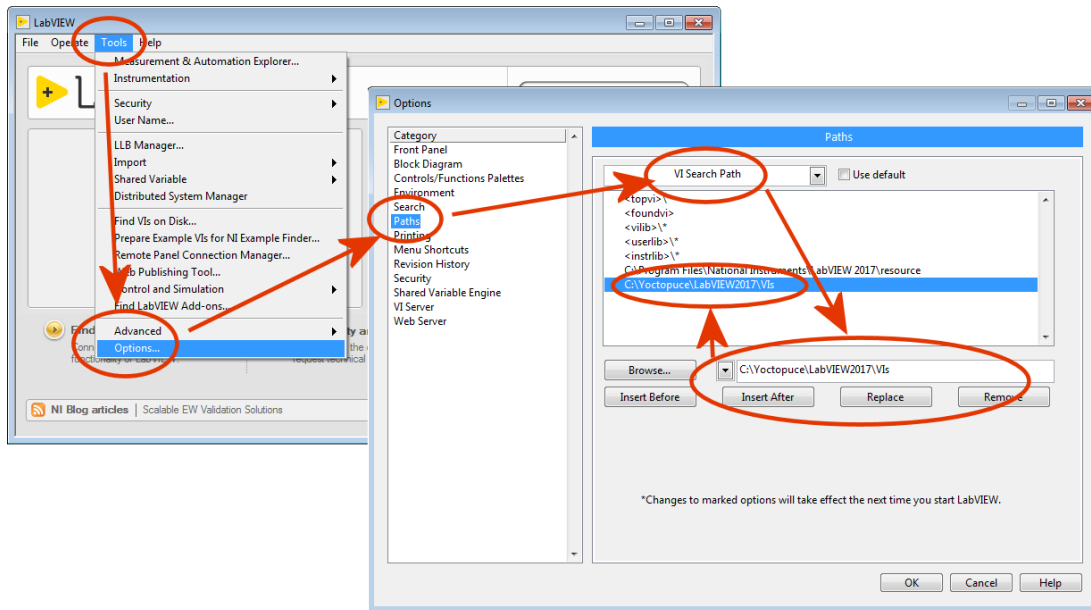
To install the LabVIEW Yoctopuce API, there are several methods.

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.

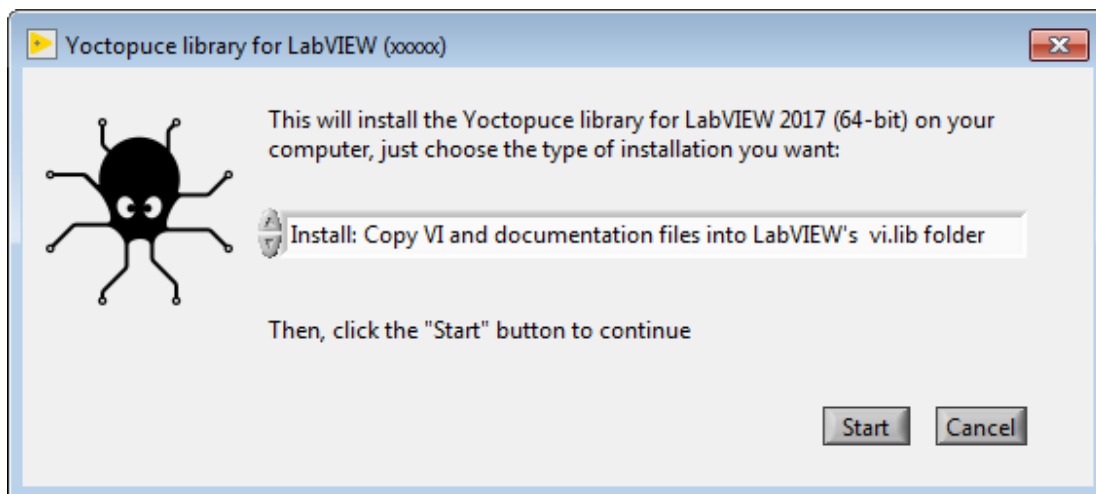
² <https://knowledge.ni.com/KnowledgeArticleDetails?id=kA00Z000000P8XnSAK>



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.

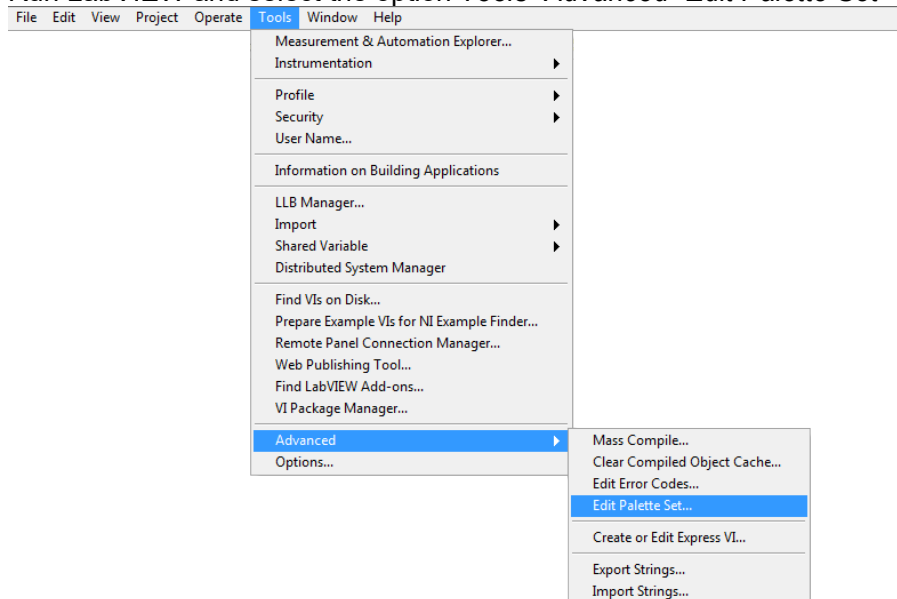
The installer identifies Yoctopuce VIs library folders by checking the presence of the *YRegisterHub.vi* file in said folders.

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:

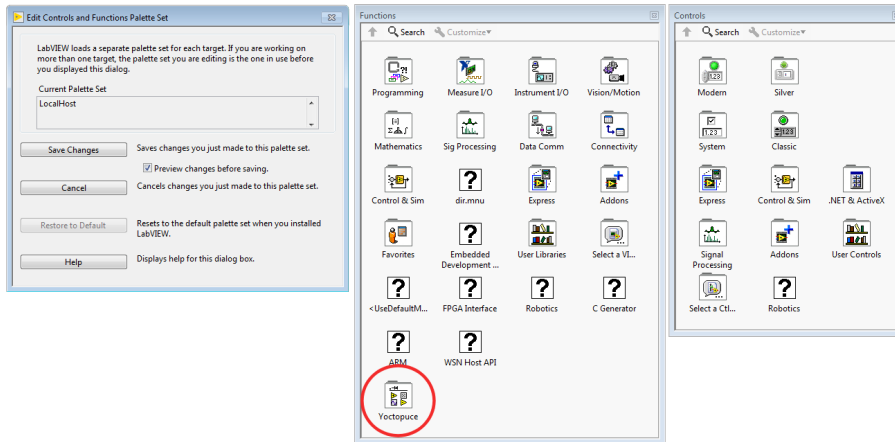
1. Create a *Yoctopuce/API* directory in the *C:\Program Files\National Instruments\LabVIEW xxxx\vi.lib* directory and copy all the VIs and DLLs of the *VIs* directory into it.
2. Create a *Yoctopuce* directory in the *C:\Program Files\National Instruments\LabVIEW xxxx\menus\Categories* directory.
3.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

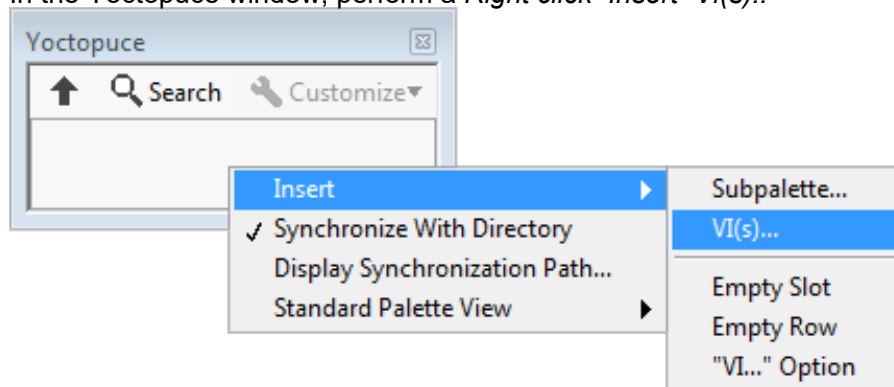
Three windows pop up:

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

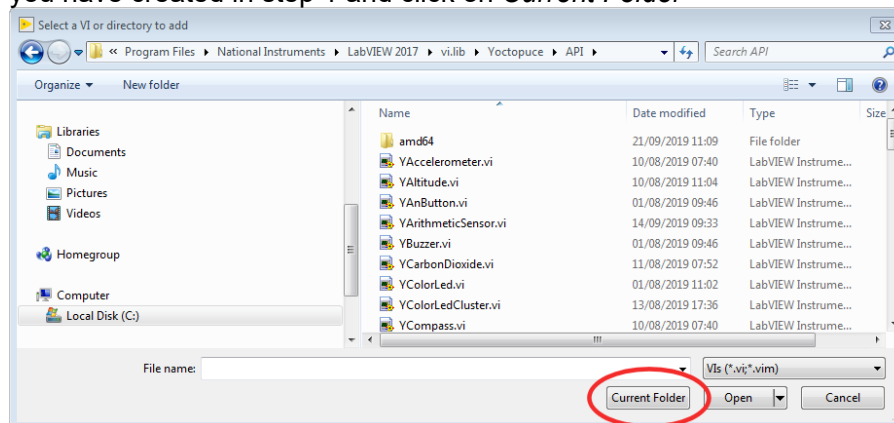


In the *Function* window, there is a *Yoctopuce* icon. Double-click it to create an empty "Yoctopuce" window.

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

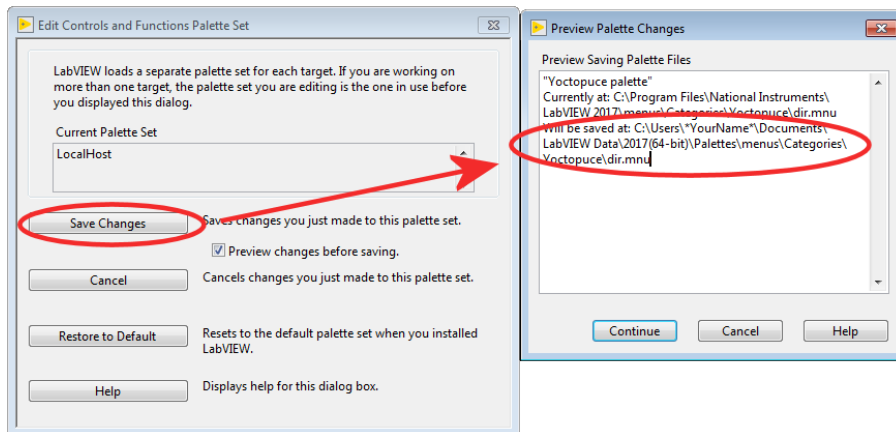


in order to open a file chooser. Put the file chooser in the *vi.lib\Yoctopuce\API* directory that you have created in step 1 and click on *Current Folder*



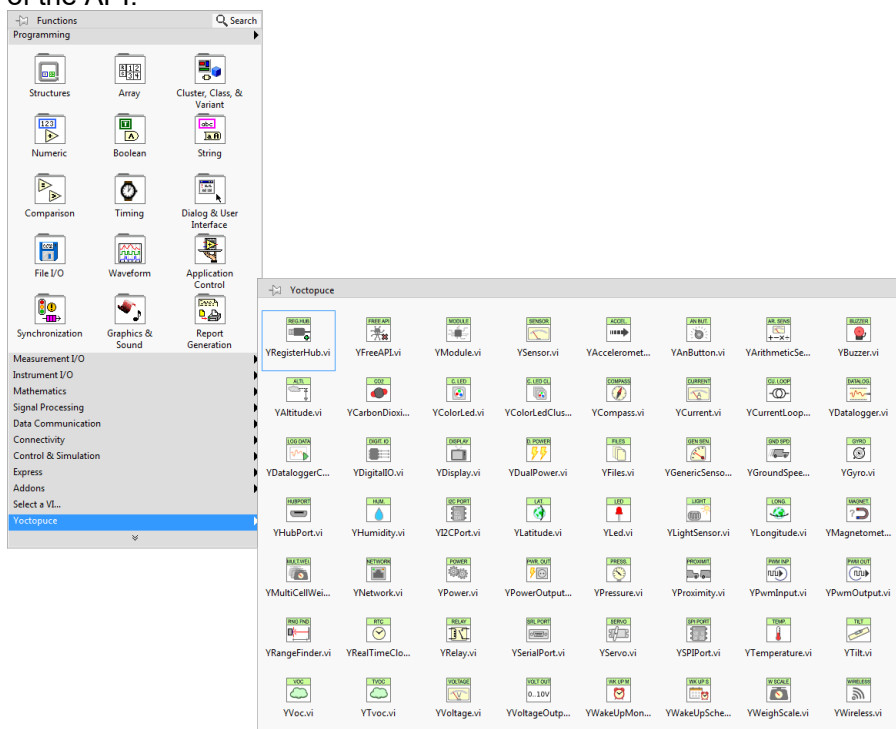
All the Yoctopuce VIs now appear in the Yoctopuce window. By default, they are sorted by alphabetical order, but you can arrange them as you see fit by moving them around with the mouse. For the palette to be easy to use, we recommend to reorganize the icons over 8 columns.

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.

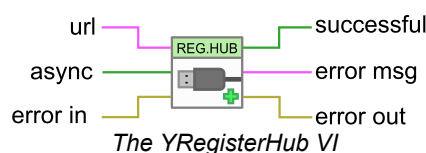


11.4. Presentation of Yoctopuce VIs

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.

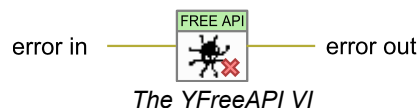
In the case of an IP address, the YRegisterHub VI tries to contact this address and generates an error if it does not succeed, unless the *async* parameter is set to TRUE. If *async* is set to TRUE, no error is generated and Yoctopuce modules corresponding to that IP address become automatically available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

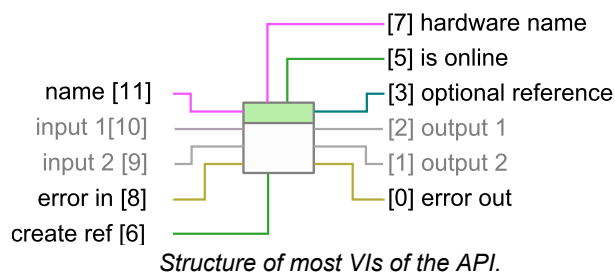
The YFreeAPI VI enables you to free resources allocated by the Yoctopuce API.



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.

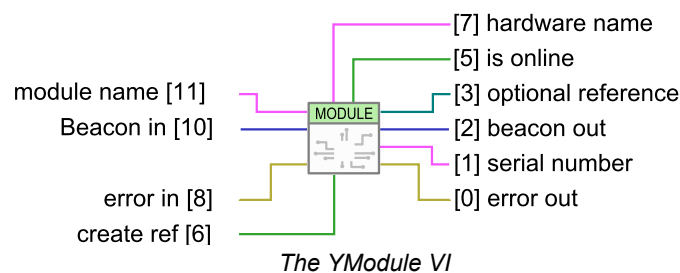
You can find the list of functions available on your Yocto-Watt in chapter *Programming, general concepts*.

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.

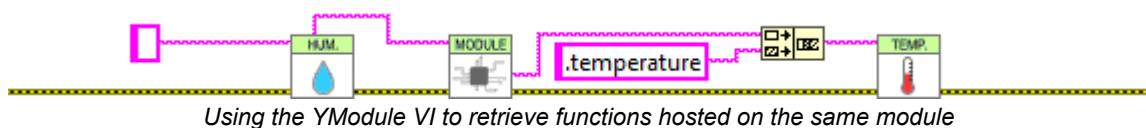
If the *name* parameter contains an empty string, the VI targets the first available function of the same type. If no function is available, *is online* is set to FALSE.

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.

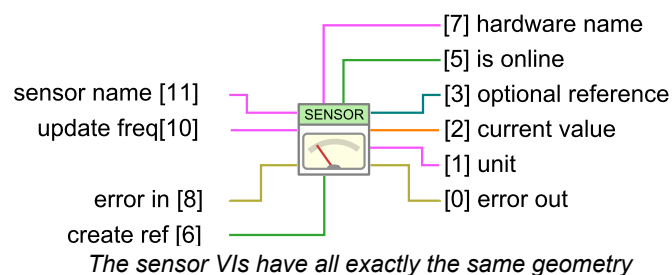


The *name* input works slightly differently from other VIs. If it is called with a *name* parameter corresponding to a function name, the `YModule` VI finds the *Module* function of the module hosting the function. You can therefore easily find the serial number of the module of any function. This enables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

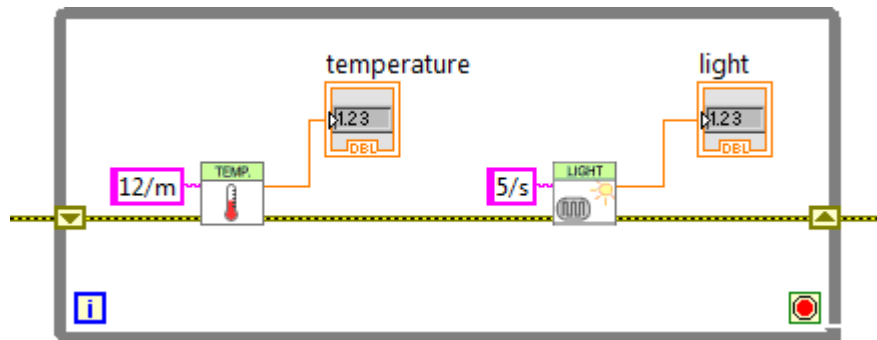


The *update freq* input parameter is a character string enabling you to configure the way in which the output value is updated:

- "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.

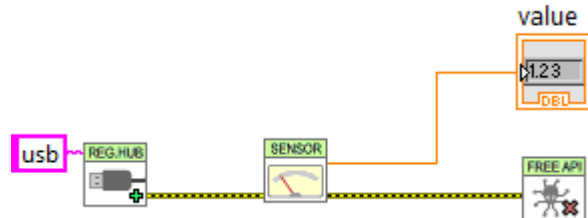


Changing the update frequency of the same module

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.

11.5. Functioning and use of VIs

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

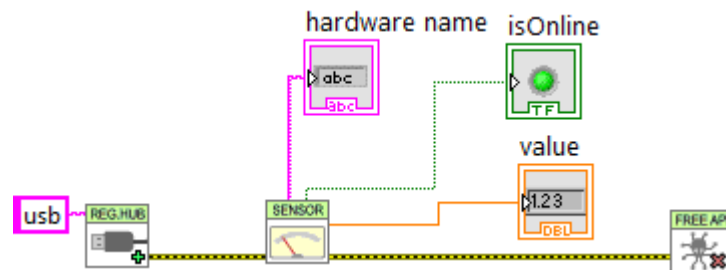


Minimal example of use of the LabVIEW Yoctopuce API

This example is based on the `YSensor` VI which is a generic VI enabling you to interface any sensor function of a Yoctopuce module. You can replace this VI by any other from the Yoctopuce API, they all have the same geometry and work in the same way. This example is limited to three actions:

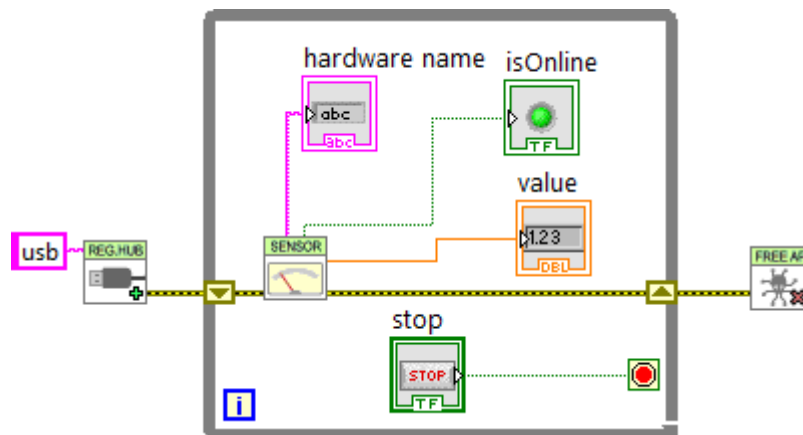
1. It initializes the API in native ("usb") mode with the `YRegisterHub` VI.
2. It displays the value of the first Yoctopuce sensor it finds thanks to the `YSensor` VI.
3. It frees the API thanks to the `YFreeAPI` VI.

This example automatically looks for an available sensor. If there is such a sensor, we can retrieve its name through the *hardware name* output and the *isOnline* output equals TRUE. If there is no available sensor, the VI does not generate an error but emulates a ghost sensor which is "offline". However, if later in the life of the application, a sensor becomes available because it has been connected, *isOnline* switches to TRUE and the *hardware name* contains the name of the sensor. We can therefore easily add a few indicators in the previous example to know how the executions goes.



Use of the hardware name and isOnline outputs

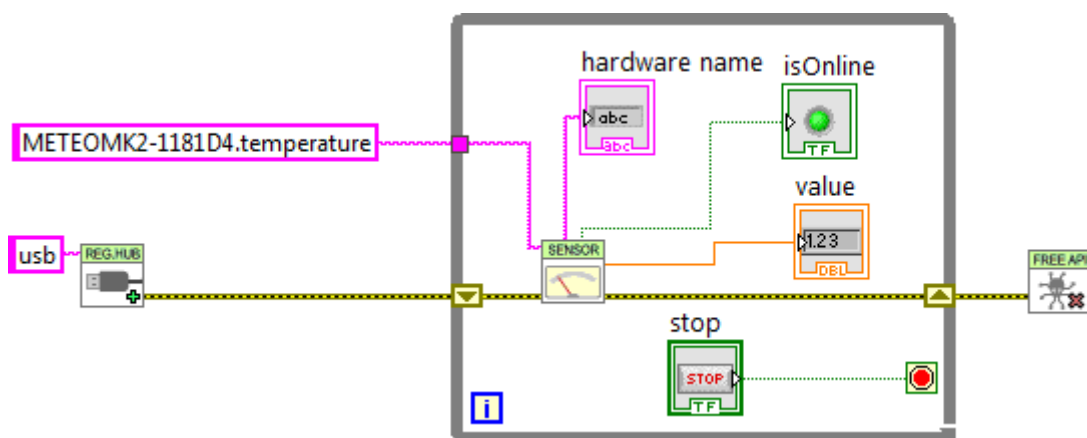
The VIs of the Yoctopuce API are actually an entry door into the library. Internally, this mechanism works independently of the Yoctopuce VIs. Indeed, most communications with electronic modules are managed automatically as background tasks. Therefore, you do not necessarily need to take any specific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



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

Note that the YRegisterHub VI is not inside the loop. The YRegisterHub VI is used to initialize the API. Unless you have several URLs that you need to register, it is better to call the YRegisterHub VI only once.

When the *name* parameter is initialized to an empty string, the Yoctopuce VIs automatically look for a function they can work with. This is very handy when you know that there is only one function of the same type available and when you do not want to manage its name. If the *name* parameter contains a hardware name or a logical name, the VI looks for the corresponding function. If it does not find it, it emulates an *offline* function while it waits for the true function to become available.

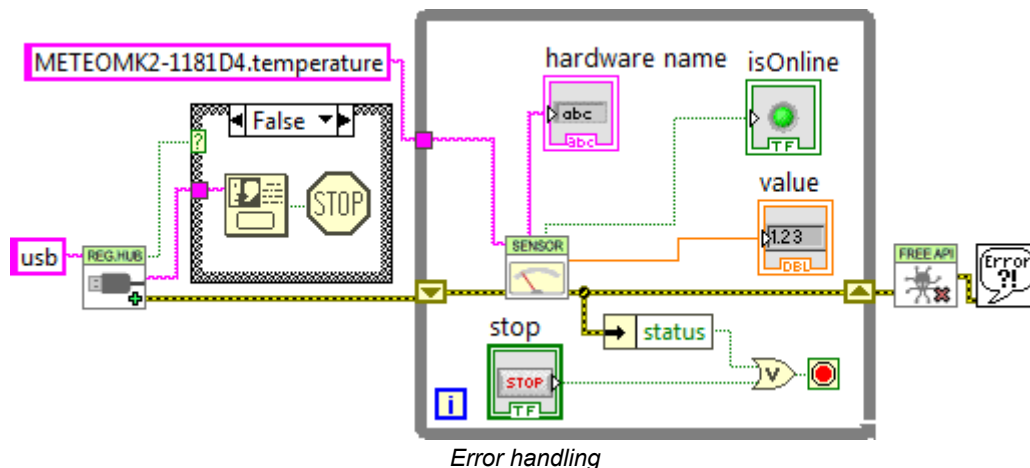


Using names to identify the functions to be used

Error handling

The LabVIEW Yoctopuce API is coded to handle errors as smoothly as possible: for example, if you use a VI to access a function which does not exist, the *isOnline* output is set to FALSE, the other outputs are set to *NaN*, and thus the inputs do not have any impact. Fatal errors are propagated through the traditional *error in*, *error out* channel.

However, the YRegisterHub VI manages connection errors slightly differently. In order to make them easier to manage, connection errors are signaled with *Success* and *error msg* outputs. If there is an issue during a call to the YRegisterHub VI, *Success* contains FALSE and *error msg* contains a description of the error.

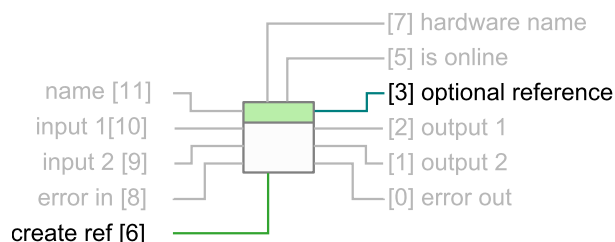


The most common error message is "Another process is already using yAPI". It means that another application, LabVIEW or other, already uses the API in native USB mode. For technical reasons, the native USB API can be used by only one application at the same time on the same machine. You can easily work around this limitation by using the network mode.

11.6. Using Proxy objects

The Yoctopuce API contains hundreds of methods, functions, and properties. It was not possible, or desirable, to create a VI for each of them. Therefore, there is a VI per class that shows the two properties that Yoctopuce deemed the most useful, but this does not mean that the rest is not available.

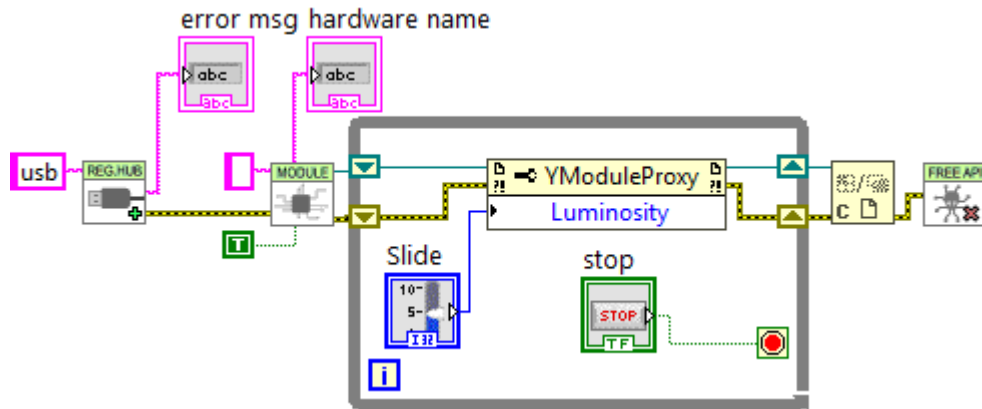
Each VI corresponding to a class has two connectors *create ref* and *optional ref* which enable you to obtain a reference on the Proxy object of the .NET Proxy API on which the LabVIEW library is built.



The connectors to obtain a reference on the Proxy object corresponding to the VI

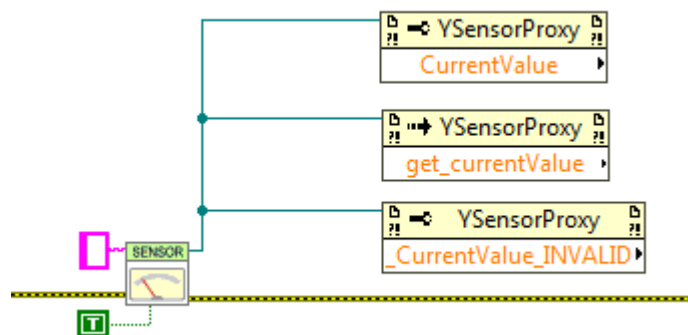
To obtain this reference, you only need to set *optional ref* to TRUE. Note, it is essential to close all references created in this way, otherwise you risk to quickly saturate the computer memory.

Here is an example which uses this technique to change the luminosity of the leds of a Yoctopuce module.



Regulating the luminosity of the leds of a module

Note that each reference allows you to obtain properties (*property nodes*) as well as methods (*invoke nodes*). By convention, properties are optimized to generate a minimum of communication with the modules. Therefore, we recommend to use them rather than the corresponding *get_xxx* and *set_xxx* methods which might seem equivalent but which are not optimized. Properties also enable you to retrieve the various constants of the API, prefixed with the "_" character. For technical reasons, the *get_xxx* and *set_xxx* methods are not all available as properties.

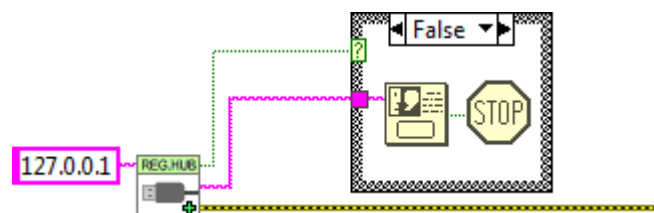


Property and Invoke nodes: Using properties, methods and constants

You can find a description of all the available properties, functions, and methods in the documentation of the *.NET Proxy API*.

Network mode

On a given machine, there can be only one process accessing local Yoctopuce modules directly by USB (url set to "usb"). It is however possible that multiple process connect in parallel to 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.

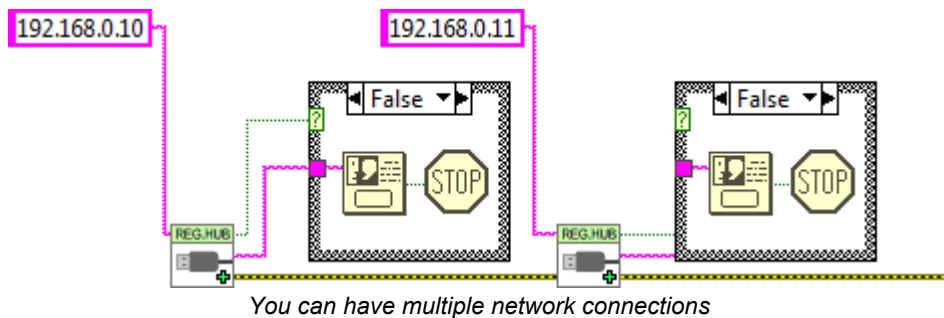


Network mode

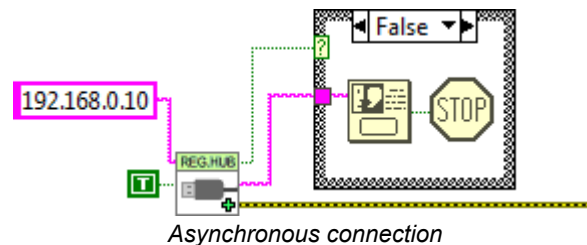
⁷ <https://www.yoctopuce.com/EN/products/category/extensions-and-networking>

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.

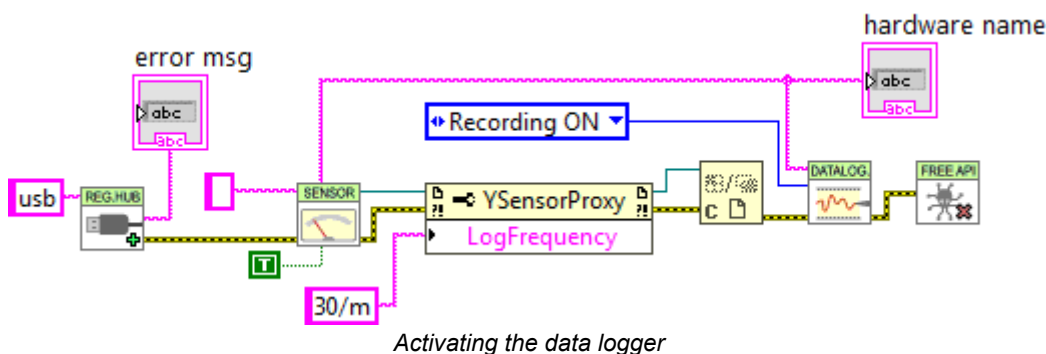


11.7. Managing the data logger

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.

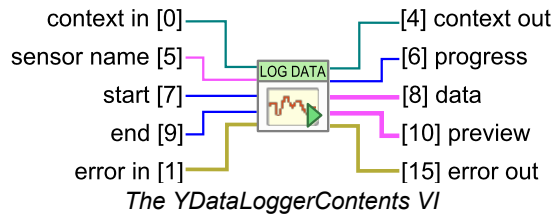
Logging

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

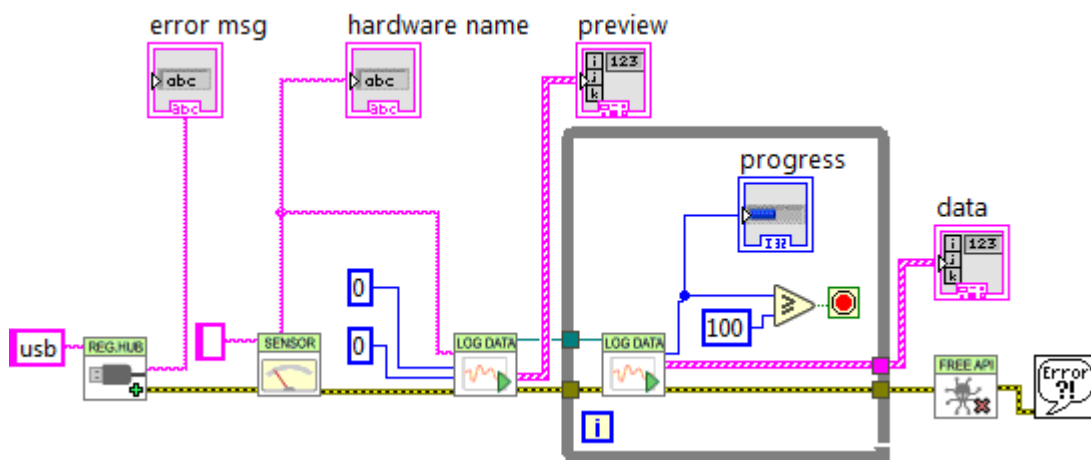
You can retrieve the data in the data logger with the YDataLoggerContents VI.



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.



Retrieving the content of the data logger

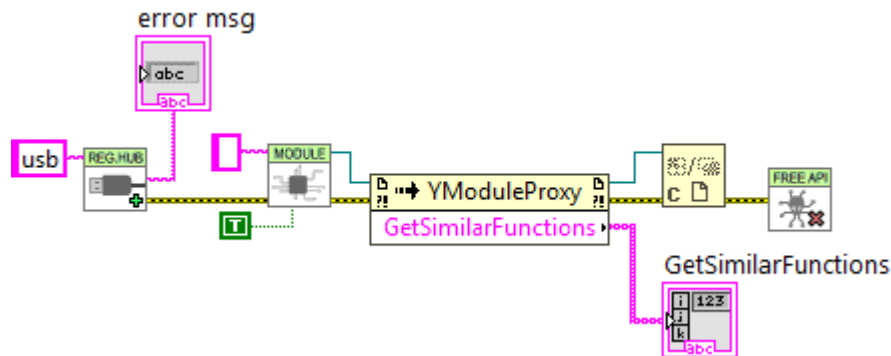
The results and the summary are returned as an array of structures containing the following fields:

- *startTime*: beginning of the measuring period
- *endTime*: end of the measuring period
- *averageValue*: average value for the period
- *minValue*: minimum value over the period
- *maxValue*: maximum value over the period

Note that if the logging frequency is superior to 1Hz, the data logger stores only current values. In this case, *averageValue*, *minValue*, and *maxValue* share the same value.

11.8. Function list

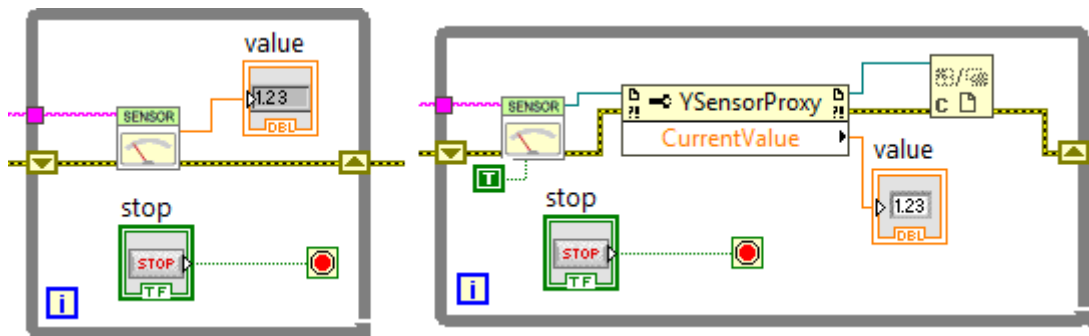
Each VI corresponding to an object of the *Proxy API* enables you to list all the functions of the same class with the *getSimilarFunctions()* method of the corresponding *Proxy* object. Thus, you can easily perform an inventory of all the connected modules, of all the connected sensors, of all the connected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



Retrieving the list of all the modules which are connected

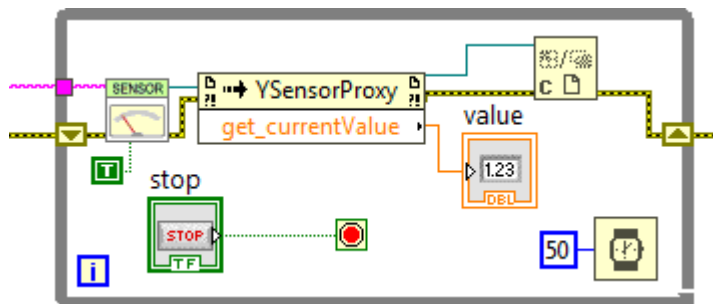
11.9. A word on performances

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

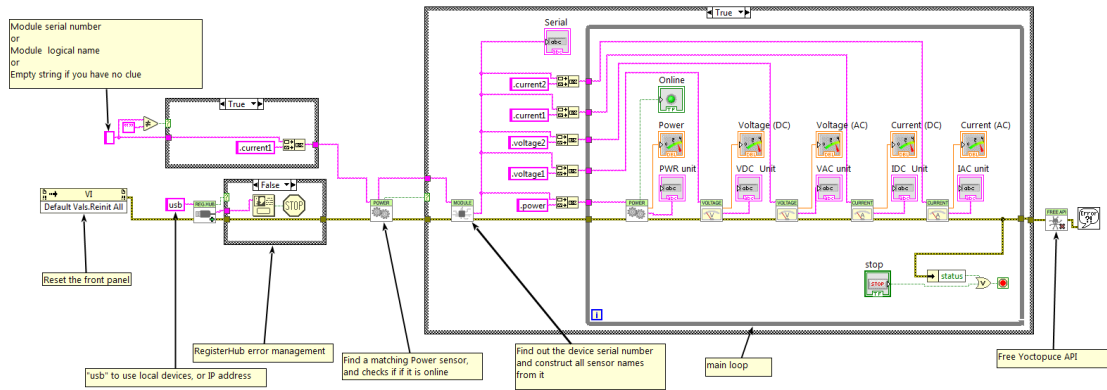
However, almost all the methods of the available Proxy objects initiate a communication with the Yoctopuce modules each time they are called. You should therefore avoid calling them too often without purpose.



This loop, using a method, must be slowed down

11.10. A full example of a LabVIEW program

Here is a short example of how to use the Yocto-Watt in LabVIEW. After a call to the *RegisterHub* VI, the *YPower* VI finds the first power sensor available, then use the *YModule* VI to find out the device serial number. This number is used to build the name of all sensors present on the device. These names are used to initialize one VI per sensor. This technique avoids ambiguities when several Yocto-Watt are connected at the same time. Once every VI is initialized, the sensor value can be displayed. When the application is about to exit, it frees the Yoctopuce API, thanks to the *YFreeAPI* VI.



Example of Yocto-Watt 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.

11.11. 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.

12. Using the Yocto-Watt with Java

Java is an object oriented language created by Sun Microsystem. Beside being free, its main strength is its portability. Unfortunately, this portability has an excruciating price. In Java, hardware abstraction is so high that it is almost impossible to work directly with the hardware. Therefore, the Yoctopuce API does not support native mode in regular Java. The Java API needs a Virtual Hub to communicate with Yoctopuce devices.

12.1. Getting ready

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

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

The library is available as source files as well as a *jar* file. Decompress the library files in a folder of your choice, connect your modules, run the VirtualHub software, and you are ready to start your first tests. You do not need to install any driver.

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

12.2. Control of the Power function

A few lines of code are enough to use a Yocto-Watt. Here is the skeleton of a Java code snippet to use the Power 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
power = YPower.FindPower("YWATMK1-123456.power");

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

¹ www.yoctopuce.com/EN/libraries.php

² www.yoctopuce.com/EN/virtualhub.php

```
// Use power.get_currentValue()
[...]
```

Let us look at these lines in more details.

YAPI.RegisterHub

The `yAPI.RegisterHub` function initializes the Yoctopuce API and indicates where the modules should be looked for. The parameter is the address of the Virtual Hub able to see the devices. If the initialization does not succeed, an exception is thrown.

YPower.FindPower

The `YPower.FindPower` function allows you to find a electrical power sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-Watt module with serial number `YWATTMK1-123456` which you have named `"MyModule"`, and for which you have given the `power` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
power = YPower.FindPower("YWATTMK1-123456.power")
power = YPower.FindPower("YWATTMK1-123456.MyFunction")
power = YPower.FindPower("MyModule.power")
power = YPower.FindPower("MyModule.MyFunction")
power = YPower.FindPower("MyFunction")
```

`YPower.FindPower` returns an object which you can then use at will to control the electrical power sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `YPower.FindPower` allows you to obtain the power consumption, as measured by the sensors. The return value is a number directly representing the value in Watts.

A real example

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

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

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

public class Demo {

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

        if (args.length == 0) {
            psensor = YPower.FirstPower();
        }
    }
}
```

```

        if (psensor == null) {
            System.out.println("No module connected (check USB cable)");
            System.exit(1);
        }
    } else {
        psensor = YPower.FindPower(args[0] + ".power");
    }

    while (true) {
        try {
            System.out.println("Current power: " + psensor.get_currentValue() + " W");
            System.out.println("  (press Ctrl-C to exit)");
            YAPI.Sleep(1000);
        } catch (YAPI_Exception ex) {
            System.out.println("Module not connected (check identification and USB
cable)");
            break;
        }
    }

    YAPI.FreeAPI();
}

```

12.3. 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();
    }
}

```

12.4. 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.

13. Using the Yocto-Watt 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.

13.1. Native access and VirtualHub

In the opposite to the classical Java API, the Java for Android API can access USB modules natively. However, as there is no VirtualHub running under Android, it is not possible to remotely control Yoctopuce modules connected to a machine under Android. Naturally, the Java for Android API remains perfectly able to connect itself to a VirtualHub running on another OS.

13.2. 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.

13.3. Compatibility

In an ideal world, you would only need to have a smart phone running under Android to be able to make Yoctopuce modules work. Unfortunately, it is not quite so in the real world. A machine running under Android must fulfil to a few requirements to be able to manage Yoctopuce USB modules natively.

¹ www.yoctopuce.com/EN/libraries.php

Android 4.x

Android 4.0 (api 14) and following are officially supported. Theoretically, support of USB *host* functions since Android 3.1. But be aware that the Yoctopuce Java for Android API is regularly tested only from Android 4 onwards.

USB *host* support

Naturally, not only must your machine have a USB port, this port must also be able to run in *host* mode. In *host* mode, the machine literally takes control of the devices which are connected to it. The USB ports of a desktop computer, for example, work in *host* mode. The opposite of the *host* mode is the *device* mode. USB keys, for instance, work in *device* mode: they must be controlled by a *host*. Some USB ports are able to work in both modes, they are *OTG (On The Go)* ports. It so happens that many mobile devices can only work in *device* mode: they are designed to be connected to a charger or a desktop computer, and nothing else. It is therefore highly recommended to pay careful attention to the technical specifications of a product working under Android before hoping to make Yoctopuce modules work with it.

Unfortunately, having a correct version of Android and USB ports working in *host* mode is not enough to guaranty that Yoctopuce modules will work well under Android. Indeed, some manufacturers configure their Android image so that devices other than keyboard and mass storage are ignored, and this configuration is hard to detect. As things currently stand, the best way to know if a given Android machine works with Yoctopuce modules consists in trying.

Supported hardware

The library is tested and validated on the following machines:

- Samsung Galaxy S3
- Samsung Galaxy Note 2
- Google Nexus 5
- Google Nexus 7
- Acer Iconia Tab A200
- Asus Tranformer Pad TF300T
- Kurio 7

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

13.4. Activating the USB port under Android

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

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

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

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

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

it stays connected. To enable the Yoctopuce library to correctly manage these authorizations, you must provide a pointer on the application context by calling the `EnableUSBHost` method of the `YAPI` class before the first USB access. This function takes as arguments an object of the `android.content.Context` class (or of a subclass). As the `Activity` class is a subclass of `Context`, it is simpler to call `YAPI.EnableUSBHost(this)` ; in the method `onCreate` of your application. If the object passed as parameter is not of the correct type, a `YAPI_Exception` exception is generated.

```
...
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    try {
        // Pass the application Context to the Yoctopuce Library
        YAPI.EnableUSBHost(this);
    } catch (YAPI_Exception e) {
        Log.e("Yocto", e.getLocalizedMessage());
    }
}
...
```

Autorun

It is possible to register your application as a default application for a USB module. In this case, as soon as a module is connected to the system, the application is automatically launched. You must add `<action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"/>` in the section `<intent-filter>` of the main activity. The section `<activity>` must have a pointer to an XML file containing the list of USB modules which can run the application.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
...
<uses-feature android:name="android.hardware.usb.host" />
...
<application ... >
    <activity
        android:name=".MainActivity" >
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>

        <meta-data
            android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"
            android:resource="@xml/device_filter" />
        </activity>
    </application>
</manifest>
```

The XML file containing the list of modules allowed to run the application must be saved in the `res/xml` directory. This file contains a list of USB *vendorId* and *deviceId* in decimal. The following example runs the application as soon as a Yocto-Relay or a YoctoPowerRelay is connected. You can find the *vendorId* and the *deviceId* of Yoctopuce modules in the characteristics section of the documentation.

```
<?xml version="1.0" encoding="utf-8"?>

<resources>
    <usb-device vendor-id="9440" product-id="12" />
    <usb-device vendor-id="9440" product-id="13" />
</resources>
```

13.5. Control of the Power function

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

```
[...]
// Enable detection of USB devices
YAPI.EnableUSBHost(this);
YAPI.RegisterHub("usb");
[...]
// Retrieve the object used to interact with the device
power = YPower.FindPower("YWATMK1-123456.power");

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

[...]
```

Let us look at these lines in more details.

YAPI.EnableUSBHost

The `YAPI.EnableUSBHost` function initializes the API with the Context of the current application. This function takes as argument an object of the `android.content.Context` class (or of a subclass). If you intend to connect your application only to other machines through the network, this function is facultative.

YAPI.RegisterHub

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

YPower.FindPower

The `YPower.FindPower` function allows you to find a electrical power sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-Watt module with serial number `YWATMK1-123456` which you have named "MyModule", and for which you have given the *power* function the name "MyFunction". The following five calls are strictly equivalent, as long as "MyFunction" is defined only once.

```
power = YPower.FindPower("YWATMK1-123456.power")
power = YPower.FindPower("YWATMK1-123456.MyFunction")
power = YPower.FindPower("MyModule.power")
power = YPower.FindPower("MyModule.MyFunction")
power = YPower.FindPower("MyFunction")
```

`YPower.FindPower` returns an object which you can then use at will to control the electrical power sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `YPower.FindPower` allows you to obtain the power consumption, as measured by the sensors. The return value is a number directly representing the value in Watts.

A real example

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

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

```
package com.yoctopuce.doc_examples;

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

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

public class GettingStarted_Yocto_Watt extends Activity implements OnItemClickListener
{
    private ArrayAdapter<String> aa;
    private String serial = "";
    private Handler handler = null;

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

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

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

```

@Override
public void onItemClick(AdapterView<?> parent, View view, int pos, long id)
{
    serial = parent.getItemAtPosition(pos).toString();
}

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

final Runnable r = new Runnable()
{
    public void run()
    {
        if (serial != null) {
            YPower power_sensor = YPower.FindPower(serial + ".power");
            try {
                TextView view = (TextView) findViewById(R.id.powerfield);
                view.setText(String.format("%.1f %s", power_sensor.getCurrentValue(),
power_sensor.getUnit()));
            } catch (YAPI_Exception e) {
                e.printStackTrace();
            }
        }
        handler.postDelayed(this, 1000);
    }
};
}

```

13.6. Control of the module part

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

```

package com.yoctopuce.doc_examples;

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

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

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

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

    @Override
    protected void onStart()

```



```

{
    super.onStart();

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

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

private void DisplayModuleInfo()
{
    TextView field;
    if (module == null)
        return;
    try {
        field = (TextView) findViewById(R.id.serialfield);
        field.setText(module.getSerialNumber());
        field = (TextView) findViewById(R.id.logicalnamefield);
        field.setText(module.getLogLogicalName());
        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.beacons witch);
        sw.setChecked(module.getBeacon() == YModule.BEACON_ON);
        field = (TextView) findViewById(R.id.logs);
        field.setText(module.get_lastLogs());

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

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

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

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

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

    try {

```

```

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

```

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

Changing the module settings

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

```

package com.yoctopuce.doc_examples;

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

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

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

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

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

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

```

```

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

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

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

@Override
public void 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();
    }
}
```

13.7. Error handling

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

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

In the 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.

14. Using Yocto-Watt 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`.

14.1. 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 (typically 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, Mac OS X, 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.

14.2. 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.

14.3. Control of the Power function

A few lines of code are enough to use a Yocto-Watt. Here is the skeleton of a TypeScript code snippet to use the Power 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 { YPower } from 'yoctolib-cjs/yocto_power.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 power: YPower = YPower.FindPower("YWATTMK1-123456.power");
```

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

```
if(await power.isOnline())
{
    // Use power.get_currentValue()
    [...]
```

Let us look at these lines in more details.

yocto_api and yocto_power import

These two imports provide access to functions allowing you to manage Yoctopuce modules. `yocto_api` is always needed, `yocto_power` is necessary to manage modules containing an electrical power sensor, such as Yocto-Watt. 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.

YPower.FindPower

The `FindPower` method allows you to find a electrical power sensor from the serial number of the module on which it resides and from its function name. You can also use logical names, as long as you have initialized them. Let us imagine a Yocto-Watt module with serial number `YWATTMK1-123456` which you have named `"MyModule"`, and for which you have given the `power` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
power = YPower.FindPower("YWATTMK1-123456.power")
power = YPower.FindPower("YWATTMK1-123456.MaFonction")
power = YPower.FindPower("MonModule.power")
power = YPower.FindPower("MonModule.MaFonction")
power = YPower.FindPower("MaFonction")
```

`YPower.FindPower` returns an object which you can then use at will to control the electrical power sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `YPower.FindPower` allows you to obtain the power consumption, as measured by the sensors. The return value is a number directly representing the value in Watts.

A real example, for Node.js

Open a command window (a terminal, a shell...) and go into the directory **example_nodejs/Doc-GettingStarted-Yocto-Watt** 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-Watt 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-Watt is connected and where you run the VirtualHub.

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

let power: YPower;

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 = YPower.FirstPower();
        if(anysensor) {
            let module: YModule = await anysensor.get_module();
            serial = await module.get_serialNumber();
        } else {
```

```

        console.log('No matching sensor connected, check cable !');
        await YAPI.FreeAPI();
        return;
    }
}
console.log('Using device '+serial);
power = YPower.FindPower(serial+".power");

refresh();
}

async function refresh(): Promise<void>
{
    if (await power.isOnline()) {
        console.log('Power : '+ (await power.get_currentValue()) + (await power.get_unit()));
    } 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-Watt**. 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.

14.4. Control of the module part

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

```
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();

```

14.5. Error handling

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

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

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

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

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

15. Using Yocto-Watt 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.

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

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

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

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

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

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

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

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

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

15.2. Using Yoctopuce library for JavaScript / EcmaScript 2017

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

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

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

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

Using the official Yoctopuce library for node.js

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

¹ www.yoctopuce.com/EN/libraries.php

² www.yoctopuce.com/EN/virtualhub.php

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

```
npm install
```

Once done, you can start the example file using:

```
node demo.js
```

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

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

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

Using the Yoctopuce library within a browser (HTML)

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

Using the Yoctoluce library on older JavaScript engines

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

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

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

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

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

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

Backward-compatibility with the old JavaScript library

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

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

by

³ <http://babeljs.io>

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

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

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

by

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

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

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

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

you can use:

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

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

You can also rewrite this last asynchronous call as:

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

15.3. Control of the Power function

A few lines of code are enough to use a Yocto-Watt. Here is the skeleton of a JavaScript code snippet to use the Power 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_power.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 power = YPower.FindPower("YWATMK1-123456.power");

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

```

if(await power.isOnline())
{
    // Use power.get_currentValue()
    [...]
}

```

Let us look at these lines in more details.

yocto_api and yocto_power import

These two import provide access to functions allowing you to manage Yoctopuce modules. `yocto_api` is always needed, `yocto_power` is necessary to manage modules containing a electrical power sensor, such as Yocto-Watt. 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.

YPower.FindPower

The `FindPower` method allows you to find a electrical power sensor from the serial number of the module on which it resides and from its function name. You can also use logical names, as long as you have initialized them. Let us imagine a Yocto-Watt module with serial number `YWATTMK1-123456` which you have named `"MyModule"`, and for which you have given the `power` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```

power = YPower.FindPower("YWATTMK1-123456.power")
power = YPower.FindPower("YWATTMK1-123456.MaFonction")
power = YPower.FindPower("MonModule.power")
power = YPower.FindPower("MonModule.MaFonction")
power = YPower.FindPower("MaFonction")

```

`YPower.FindPower` returns an object which you can then use at will to control the electrical power sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `YPower.FindPower` allows you to obtain the power consumption, as measured by the sensors. The return value is a number directly representing the value in Watts.

A real example, for Node.js

Open a command window (a terminal, a shell...) and go into the directory **example_nodejs/Doc-GettingStarted-Yocto-Watt** 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-Watt 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-Watt is connected and where you run the VirtualHub.

```

"use strict";

```

```

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

let power;

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

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

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

    refresh();
}

async function refresh()
{
    if (await power.isOnline()) {
        console.log('Power : '+ (await power.get_currentValue()) + (await power.get_unit()));
    } 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-Watt**. 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_power.js"></script>
<script>
  async function startDemo()
  {
    await YAPI.LogUnhandledPromiseRejections();
    await YAPI.DisableExceptions();

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

  async function refresh()
  {
    let serial = document.getElementById('serial').value;
    if(serial == '') {
      // by default use any connected module suitable for the demo
      let anysensor = YPower.FirstPower();
      if(anysensor) {
        let module = await anysensor.module();
        serial = await module.get_serialNumber();
        document.getElementById('serial').value = serial;
      }
    }
    let power = YPower.FindPower(serial+".power");

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

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

```

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

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.

```

"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);
}

```

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-Watt with PHP

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

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

To start your tests with PHP, you need a PHP 5.3 (or more) server¹, preferably locally on you machine. If you wish to use the PHP server of your internet provider, it is possible, but you will probably need to configure your ADSL router for it to accept and forward TCP request on the 4444 port.

16.1. Getting ready

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

- The PHP programming library²
- The VirtualHub software³ for Windows, Mac OS X, or Linux, depending on your OS

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

16.2. Control of the Power function

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

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

¹ A couple of free PHP servers: easyPHP for Windows, MAMP for Mac OS X.

² www.yoctopuce.com/EN/libraries.php

³ www.yoctopuce.com/EN/virtualhub.php

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

// Retrieve the object used to interact with the device
$power = YPower::FindPower("YWATTMK1-123456.power");

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

Let's look at these lines in more details.

yocto_api.php and yocto_power.php

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

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.

YPower::FindPower

The `YPower::FindPower` function allows you to find a electrical power sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-Watt module with serial number `YWATTMK1-123456` which you have named "*MyModule*", and for which you have given the *power* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
$power = YPower::FindPower("YWATTMK1-123456.power");
$power = YPower::FindPower("YWATTMK1-123456.MyFunction");
$power = YPower::FindPower("MyModule.power");
$power = YPower::FindPower("MyModule.MyFunction");
$power = YPower::FindPower("MyFunction");
```

`YPower::FindPower` returns an object which you can then use at will to control the electrical power sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `yFindPower` allows you to obtain the power consumption, as measured by the sensors. The return value is a number directly representing the value in Watts.

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

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

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

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

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

  $tvalue = $powersensor->get_currentValue();
  Print("Power: $tvalue W<br>");
  YAPI::FreeAPI();

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

16.3. Control of the module part

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

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

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

  // Setup the API to use the VirtualHub on local machine
  if(YAPI::RegisterHub('http://127.0.0.1:4444/', $errmsg) != YAPI::SUCCESS) {
```

```

        die("Cannot contact VirtualHub on 127.0.0.1 : ".$errmsg);
    }

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

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

```

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

Changing the module settings

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

```

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

    // Use explicit error handling rather than exceptions

```

```

YAPI::DisableExceptions();

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

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

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

```

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

Listing the modules

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

```

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

```

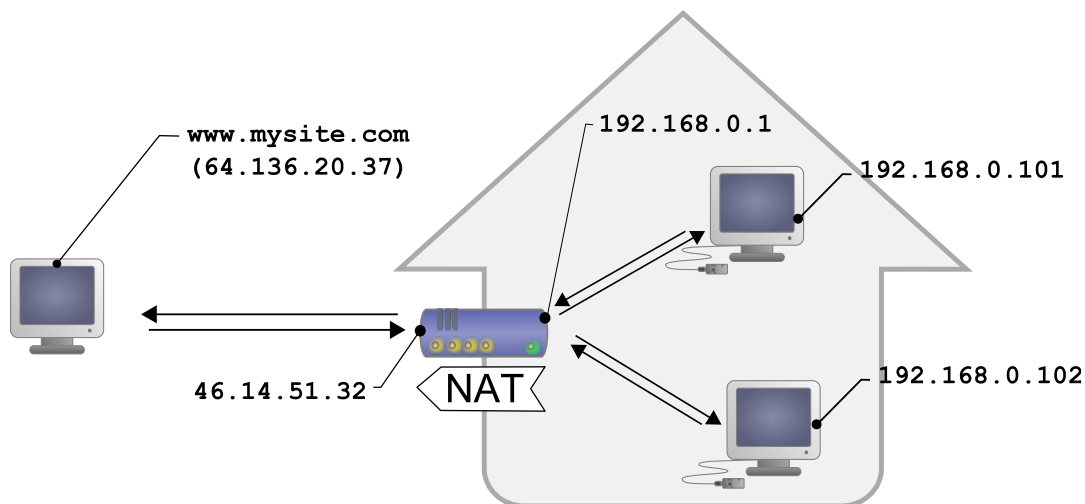
```
?>
</TT>
</BODY>
</HTML>
```

16.4. HTTP callback API and NAT filters

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

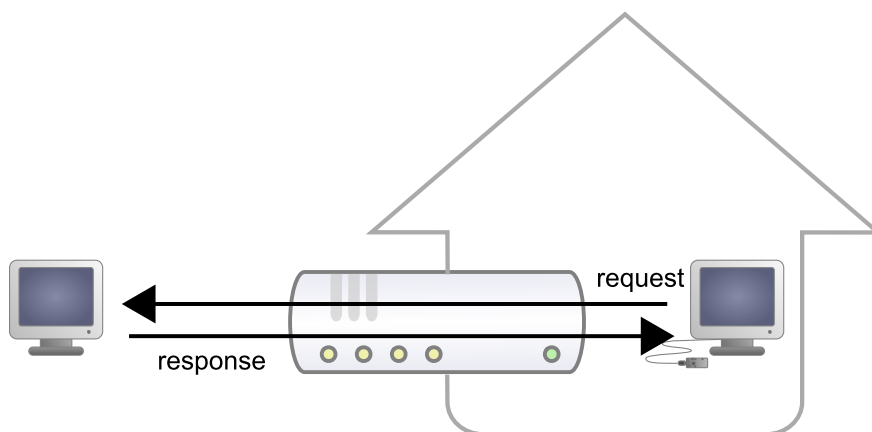
The NAT filter: advantages and disadvantages

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

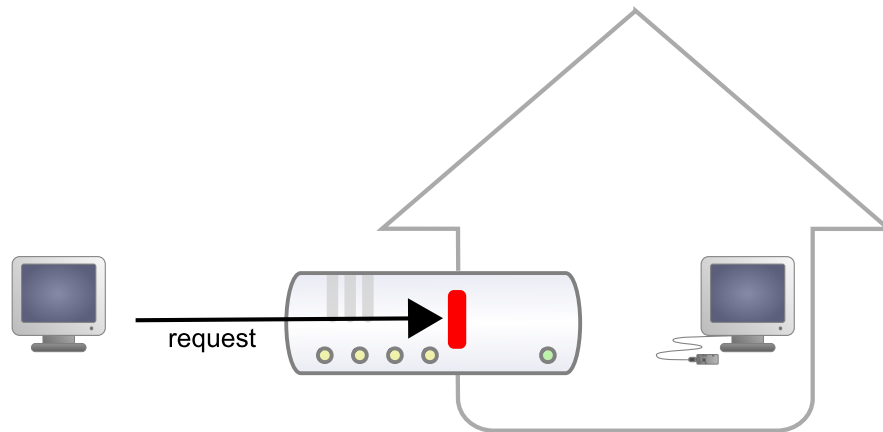


Typical DSL configuration: LAN machines are isolated from the outside by the DSL router

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



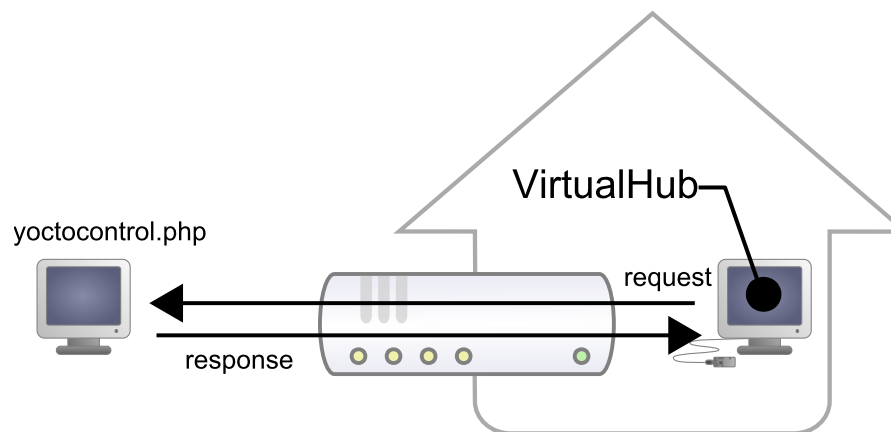
Responses from request from LAN machines are routed.



But requests from the outside are blocked.

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

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



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

Configuration

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

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

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

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

Click on the "configure" button on the first line

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

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

Incoming connections

Authentication to read information from the devices: NO
 Authentication to make changes to the devices: NO

Outgoing callbacks

Callback URL: octoHub
 Delay between callbacks: min: 3 [s] max: 600 [s]

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

Edit callback

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

- Specify the Type of callback you want to use:

Yoctopuce devices can be controlled through remote PHP scripts. That Yocto-API callback protocol is designed so it can pass through NAT filters without opening ports. See your device user manual, *PHP programming* section for more details.
- Specify the URL to use for reporting values. *HTTPS protocol is not yet supported.*
 Callback URL:
- If your callback requires authentication, enter credentials here. Digest authentication is recommended, but Basic authentication works as well.
 Username:
 Password:
- Setup the desired frequency of notifications:
 No less than seconds between two notification
 But notify after seconds in any case
- Press on the **Test** button to check your parameters.
- 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.

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-Watt with Visual Basic .NET

VisualBasic has long been the most favored entrance path to the Microsoft world. Therefore, we had to provide our library for this language, even if the new trend is shifting to C#. All the examples and the project models are tested with Microsoft VisualBasic 2010 Express, freely available on the Microsoft web site¹.

17.1. Installation

Download the Visual Basic Yoctopuce library from the Yoctopuce web site². There is no setup program, simply copy the content of the zip file into the directory of your choice. You mostly need the content of the `Sources` directory. The other directories contain the documentation and a few sample programs. All sample projects are Visual Basic 2010, projects, if you are using a previous version, you may have to recreate the projects structure from scratch.

17.2. Using the Yoctopuce API in a Visual Basic project

The Visual Basic.NET Yoctopuce library is composed of a DLL and of source files in Visual Basic. The DLL is not a .NET DLL, but a classic DLL, written in C, which manages the low level communications with the modules³. The source files in Visual Basic manage the high level part of the API. Therefore, you need both this DLL and the .vb files of the `sources` directory to create a project managing Yoctopuce modules.

Configuring a Visual Basic project

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

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

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

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

² www.yoctopuce.com/EN/libraries.php

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

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

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

17.3. Control of the Power function

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

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

' Retrieve the object used to interact with the device
Dim power As YPower
power = YPower.FindPower("YWATMK1-123456.power")

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

YPower.FindPower

The `YPower.FindPower` function allows you to find a electrical power sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-Watt module with serial number `YWATMK1-123456` which you have named `"MyModule"`, and for which you have given the `power` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
power = YPower.FindPower("YWATMK1-123456.power")
power = YPower.FindPower("YWATMK1-123456.MyFunction")
power = YPower.FindPower("MyModule.power")
power = YPower.FindPower("MyModule.MyFunction")
power = YPower.FindPower("MyFunction")
```

`YPower.FindPower` returns an object which you can then use at will to control the electrical power sensor.

isOnline

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

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

get_currentValue

The `get_currentValue()` method of the object returned by `yFindPower` allows you to obtain the power consumption, as measured by the sensors. The return value is a number directly representing the value in Watts.

A real example

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

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

```
Module Module1

    Private Sub Usage()

        Dim execname = System.AppDomain.CurrentDomain.FriendlyName
        Console.WriteLine("Usage:")
        Console.WriteLine(execname + " <serial_number>")
        Console.WriteLine(execname + " <logical_name>")
        Console.WriteLine(execname + " any ")
        System.Threading.Thread.Sleep(2500)
    End Sub
End Sub

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

    Dim psensor As YPower

    If argv.Length < 2 Then Usage()

    target = argv(1)

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

    If target = "any" Then
        psensor = YPower.FirstPower()
        If psensor Is Nothing Then
            Console.WriteLine("No module connected (check USB cable) ")
        End If
    End If
    Else
        psensor = YPower.FindPower(target + ".power")
    End If

    While (True)
        If Not (psensor.IsOnline()) Then
            Console.WriteLine("Module not connected (check identification and USB cable)")
        End If
        Console.WriteLine("Current Power: " + Str(psensor.get_currentValue()) + " W")
        Console.WriteLine(" (press Ctrl-C to exit)")
        YAPI.Sleep(1000, errmsg)
    End While
    YAPI.FreeAPI()

End Sub

End Module
```

17.4. Control of the module part

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

```
Imports System.IO
Imports System.Environment

Module Module1

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

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

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

If argv.Length < 2 Then usage()

m = YModule.FindModule(argv(1)) REM use serial or logical name
If (m.isOnline()) Then
    If argv.Length > 2 Then
        If argv(2) = "ON" Then m.set_beacon(Y_BEACON_ON)
        If argv(2) = "OFF" Then m.set_beacon(Y_BEACON_OFF)
    End If
    Console.WriteLine("serial:      " + m.get_serialNumber())
    Console.WriteLine("logical name: " + m.get_logicalName())
    Console.WriteLine("luminosity:   " + Str(m.get_luminosity()))
    Console.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
End Sub

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

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

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

    m = YModule.FindModule(argv(1)) REM use serial or logical name
    If m.isOnline() Then
        newname = argv(2)
        If (Not YAPI.CheckLogicalName(newname)) Then
            Console.WriteLine("Invalid name (" + newname + ")")
        End
        End If
        m.set_logicalName(newname)
        m.saveToFlash() REM do not forget this
        Console.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

```

End Module

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-Watt with Delphi

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

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

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

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

18.1. Preparation

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

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

18.2. Control of the Power function

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

```
uses yocto_api, yocto_power;

var errmsg: string;
    power: TYPower;

[...]
```

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

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

³ www.yoctopuce.com/EN/libraries.php

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

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

// Retrieve the object used to interact with the device
power = yFindPower("YWATTMK1-123456.power")

// Hot-plug is easy: just check that the device is online
if power.isOnline() then
begin
    // Use power.get_currentValue()
    [...]
end;
[...]
```

Let's look at these lines in more details.

yocto_api and yocto_power

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

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.

yFindPower

The `yFindPower` function allows you to find a electrical power sensor from the serial number of the module on which it resides and from its function name. You can also use logical names, as long as you have initialized them. Let us imagine a Yocto-Watt module with serial number `YWATTMK1-123456` which you have named `"MyModule"`, and for which you have given the *power* function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
power := yFindPower("YWATTMK1-123456.power");
power := yFindPower("YWATTMK1-123456.MyFunction");
power := yFindPower("MyModule.power");
power := yFindPower("MyModule.MyFunction");
power := yFindPower("MyFunction");
```

`yFindPower` returns an object which you can then use at will to control the electrical power sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `yFindPower` allows you to obtain the power consumption, as measured by the sensors. The return value is a number directly representing the value in Watts.

A real example

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

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

```
program helloworld;
{$APPTYPE CONSOLE}
```

```

uses
  SysUtils,
  Windows,
  yocto_api,
  yocto_power;

Procedure Usage();
var
  exe : string;
begin
  exe:= ExtractFileName(paramstr(0));
  WriteLn(exe+' <serial_number>');
  WriteLn(exe+' <logical_name>');
  WriteLn(exe+' any');
  sleep(2500);
  halt;
End;

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

begin

  if (paramcount<1) then usage();

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

  if paramstr(1)='any' then
  begin
    // try to find the first Power sensor available
    sensor := yFirstPower();
    if sensor=nil then
    begin
      writeln('No module connected (check USB cable)');
      halt;
    end
  end
  else // or use the one specified on the commande line
    sensor:= YFindPower(paramstr(1)+'.power');

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

end.

```

18.3. Control of the module part

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

```

program modulecontrol;
{$APPTYPE CONSOLE}
uses

```

```

SysUtils,
yocto_api;

const
  serial = 'YWATTMK1-123456'; // use serial number or logical name

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

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

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

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

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

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

```

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

Changing the module settings

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

```

program saveSettings;
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Yocto_API;

const
  serial = 'YWATMK1-123456'; // use serial number or logical name

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

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

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

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

```

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

Listing the modules

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

```

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

var
  module : TYModule;
  errmsg : string;

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

```

```

Writeln('Device list');

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

end.

```

18.4. Error handling

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

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

Universal Windows Platform (UWP) is not a language per se, 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.

19.1. 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.

19.2. 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.

19.3. 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-Watt, here is what you must add in the "Capabilities" node:

```
<DeviceCapability Name="humaninterfacedevice">
  <!-- Yocto-Watt -->
  <Device Id="vidpid:24e0 002A">
    <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.

19.4. Control of the Power function

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

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

// Retrieve the object used to interact with the device
YPower power = YPower.FindPower("YWATTMK1-123456.power");

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

[...]
```

Let us look at these lines in more details.

YAPI.RegisterHub

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

YPower.FindPower

The `YPower.FindPower` function allows you to find a electrical power sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-Watt module with serial number `YWATTMK1-123456` which you have named "MyModule", and for which you have given the *power* function the name "MyFunction". The following five calls are strictly equivalent, as long as "MyFunction" is defined only once.

```
power = YPower.FindPower("YWATTMK1-123456.power");
power = YPower.FindPower("YWATTMK1-123456.MaFonction");
power = YPower.FindPower("MonModule.power");
power = YPower.FindPower("MonModule.MaFonction");
power = YPower.FindPower("MaFonction");
```

`YPower.FindPower` returns an object which you can then use at will to control the electrical power sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `YPower.FindPower` allows you to obtain the power consumption, as measured by the sensors. The return value is a number directly representing the value in Watts.

19.5. A real example

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

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

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

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

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

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

                YPower psensor;

                if (Target.ToLower() == "any") {
                    psensor = YPower.FirstPower();

                    if (psensor == null) {
                        WriteLine("No module connected (check USB cable) ");
                        return -1;
                    }
                } else {
                    psensor = YPower.FindPower(Target + ".power");
                }

                while (await psensor.isOnline()) {
                    WriteLine("Current power: " + await psensor.get_currentValue() + " W");
                    await YAPI.Sleep(1000);
                }

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

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

19.6. Control of the module part

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

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

```

using com.yoctopuce.YoctoAPI;

namespace Demo
{
    public class Demo : DemoBase
    {

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

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

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

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

```

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

Changing the module settings

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

```

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

namespace Demo
{
    public class Demo : DemoBase
    {

        public string HubURL { get; set; }

```

```

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

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

        await YAPI.RegisterHub(HubURL);

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

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

```

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

Listing the modules

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

```

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

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

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

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

```

```

        return 0;
    }
}

```

19.7. Error handling

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

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

In the 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);
}

```


20. Using Yocto-Watt with Objective-C

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

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

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

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

20.1. Control of the Power function

A few lines of code are enough to use a Yocto-Watt. Here is the skeleton of a Objective-C code snippet to use the Power function.

```
#import "yocto_api.h"
#import "yocto_power.h"

...
NSError *error;
[YAPI RegisterHub:@"usb": &error]
...
// On récupère l'objet représentant le module (ici connecté en local sur USB)
power = [YPower FindPower:@"YWATTMK1-123456.power"];

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

¹ www.yoctopuce.com/EN/libraries.php

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

```
// Utiliser [power get_currentValue]
...
}
```

Let's look at these lines in more details.

yocto_api.h and yocto_power.h

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

[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.

[Power FindPower]

The `[Power FindPower]` function allows you to find a electrical power sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-Watt module with serial number `YWATTMK1-123456` which you have named `"MyModule"`, and for which you have given the `power` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
YPower *power = [Power FindPower:@"YWATTMK1-123456.power"];
YPower *power = [Power FindPower:@"YWATTMK1-123456.MyFunction"];
YPower *power = [Power FindPower:@"MyModule.power"];
YPower *power = [Power FindPower:@"MyModule.MyFunction"];
YPower *power = [Power FindPower:@"MyFunction"];
```

`[Power FindPower]` returns an object which you can then use at will to control the electrical power sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `YPower.FindPower` allows you to obtain the power consumption, as measured by the sensors. The return value is a number directly representing the value in Watts.

A real example

Launch Xcode 4.2 and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-Watt** 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_power.h"

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



```

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

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

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

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

            NSLog(@"Current Power: %f W\n", [psensor get_currentValue]);
            NSLog(@"      (press Ctrl-C to exit)\n");
            [YAPI Sleep:1000:NULL];
        }
        [YAPI FreeAPI];
    }
    return 0;
}

```

20.2. Control of the module part

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

```

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

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

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

    @autoreleasepool {
        // Setup the API to use local USB devices
        if([YAPI RegisterHub:@"usb": &error] != YAPI_SUCCESS) {
            NSLog(@"RegisterHub error: %@", [error localizedDescription]);
            return 1;
        }
        if(argc < 2)
            usage(argv[0]);
        NSString *serial_or_name = [NSString stringWithUTF8String:argv[1]];
        // use serial or logical name
        YModule *module = [YModule FindModule:serial_or_name];
        if ([module isOnline]) {

```

```

    if (argc > 2) {
        if (strcmp(argv[2], "ON") == 0)
            [module setBeacon:Y_BEACON_ON];
        else
            [module setBeacon:Y_BEACON_OFF];
    }
    NSLog(@"serial:      %@\n", [module serialNumber]);
    NSLog(@"logical name: %@\n", [module logicalName]);
    NSLog(@"luminosity:   %d\n", [module luminosity]);
    NSLog(@"beacon:      ");
    if ([module beacon] == Y_BEACON_ON)
        NSLog(@"ON\n");
    else
        NSLog(@"OFF\n");
    NSLog(@"upTime:      %ld sec\n", [module upTime] / 1000);
    NSLog(@"USB current:  %d mA\n", [module usbCurrent]);
    NSLog(@"logs:   %@\n", [module get_lastLogs]);
} else {
    NSLog(@"%@ not connected (check identification and USB cable)\n",
          serial_or_name);
}
[YAPI FreeAPI];
}
return 0;
}

```

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

Changing the module settings

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

```

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

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

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

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

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

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

        if (module.isOnline) {
            if (argc >= 3) {
                NSString *newname = [NSString stringWithUTF8String:argv[2]];
                if (![YAPI CheckLogicalName:newname]) {
                    NSLog(@"Invalid name (%@)\n", newname);
                    usage(argv[0]);
                }
            }
        }
    }
}

```

```

    }
    module.logicalName = newname;
    [module saveToFlash];
}
NSLog(@"Current name: %@\n", module.logicalName);
} else {
    NSLog(@"%@ not connected (check identification and USB cable)\n",
          serial_or_name);
}
[YAPI FreeAPI];
}
return 0;
}

```

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

Listing the modules

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

```

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

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

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

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

        YModule *module = [YModule FirstModule];
        while (module != nil) {
            NSLog(@"%03  %03", module.serialNumber, module.productName);
            module = [module nextModule];
        }
        [YAPI FreeAPI];
    }
    return 0;
}

```

20.3. 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.

21. Using with unsupported languages

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

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

21.1. Command line

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

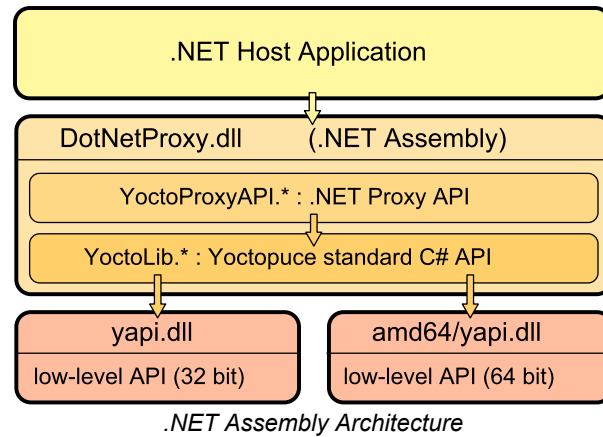
However, if the command line API is the easiest solution, it is neither the fastest nor the most efficient. For each call, the executable must initialize its own API and make an inventory of USB connected modules. This requires about one second per call.

21.2. .NET Assembly

A .NET Assembly enables you to share a set of pre-compiled classes to offer a service, by stating entry points which can be used by third-party applications. In our case, it's the whole Yoctopuce library which is available in the .NET Assembly, so that it can be used in any environment which supports .NET Assembly dynamic loading.

The Yoctopuce library as a .NET Assembly does not contain only the standard C# Yoctopuce library, as this wouldn't have allowed an optimal use in all environments. Indeed, we cannot expect host applications to necessarily offer a thread system or a callback system, although they are very useful to manage plug-and-play events and sensors with a high refresh rate. Likewise, we can't expect from external applications a transparent behavior in cases where a function call in Assembly creates a delay because of network communications.

Therefore, we added to it an additional layer, called *.NET Proxy* library. This additional layer offers an interface very similar to the standard library but somewhat simplified, as it internally manages all the callback mechanisms. Instead, this library offers mirror objects, called *Proxys*, which publish through *Properties* the main attributes of the Yoctopuce functions such as the current measure, configuration parameters, the state, and so on.



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 don't want to use the Proxy library, and if your environment allows it, you can use the standard C# API as it is located in the Assembly, under the `YoctoLib` namespace. Beware however not to mix both types of use: either you go through the Proxy library, or you use 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.

21.3. VirtualHub and HTTP GET

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

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

REST interface

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

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

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

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

Driving a module through the REST interface

Each Yoctopuce module has its own REST interface, available in several variants. Let us imagine a Yocto-Watt with the `YWATTMK1-12345` 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/YWATTMK1-12345/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/YWATTMK1-12345/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/YWATTMK1-12345/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 power function, build the following URL:

```
http://127.0.0.1:4444/bySerial/YWATTMK1-12345/api/power.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/YWATTMK1-12345/api/power/logicalName
```

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

```
http://127.0.0.1:4444/bySerial/YWATTMK1-12345/api/power?logicalName=myFunction
```

You can find the list of available attributes for your Yocto-Watt 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/YWATTMK1-12345/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/YWATTMK1-12345/dataLogger.json?id=power&utc=1389801080
```


21.4. Using dynamic libraries

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

Filename	Platform
libyapi.dylib	Mac 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 bufsize, int *fullsize,
char *errmsg);
```

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

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

Finally, the *yapiHTTPRequest* function sends HTTP requests to the module REST API. The *device* parameter contains the serial number or the logical name of the module which you want to reach. The *request* parameter contains the full HTTP request (including terminal line breaks). *buffer* points to a character buffer long enough to contain the answer. *bufsize* is the size of the buffer. *fullsize* is a pointer to an integer to which will be assigned the actual size of the answer. The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message. This pointer can also point to *null*. The function returns a negative integer in case of error, zero otherwise.

The format of the requests is the same as the one described in the *VirtualHub et HTTP GET* section. All the character strings used by the API are strings made of 8-bit characters: Unicode and UTF8 are not supported.

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

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

```
// Dll functions import
function yapiInitAPI(mode:integer;
                    errmsg : pansichar):integer; cdecl;
```

```

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

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

const
    serial        = 'YWATTMK1-12345';
    getValue = 'GET /api/module/luminosity HTTP/1.1'#13#10#13#10;
    setValue = 'GET /api/module?luminosity=100 HTTP/1.1'#13#10#13#10;

begin
    errmsg := @errmsgBuffer;
    data := @dataBuffer;
    // API initialization
    if(yapiInitAPI(1,errmsg)<0) then
        begin
            writeln(errmsg);
            halt;
        end;

    // forces a device inventory
    if( yapiUpdateDeviceList(1,errmsg)<0) then
        begin
            writeln(errmsg);
            halt;
        end;

    // requests the module luminosity
    if (yapiHTTPRequest(serial,getValue,data,sizeof(dataBuffer),fullsize,errmsg)<0) then
        begin
            writeln(errmsg);
            halt;
        end;

    // searches for the HTTP header end
    p := pos(#13#10#13#10,data);

    // displays the response minus the HTTP header
    writeln(copy(data,p+4,length(data)-p-3));

    // changes the luminosity
    if (yapiHTTPRequest(serial,setValue,data,sizeof(dataBuffer),fullsize,errmsg)<0) then
        begin
            writeln(errmsg);
            halt;
        end;

end.

```

Module inventory

To perform an inventory of Yoctopuce modules, you need two functions from the dynamic library:

```

int yapiGetAllDevices(int *buffer,int maxsize,int *neededsize,char *errmsg);
int yapiGetDeviceInfo(int devdesc,yDeviceSt *infos, char *errmsg);

```

The *yapiGetAllDevices* function retrieves the list of all connected modules as a list of handles. *buffer* points to a 32-bit integer array which contains the returned handles. *maxsize* is the size in bytes of the buffer. To *neededsize* is assigned the necessary size to store all the handles. From this, you can deduce either the number of connected modules or that the input buffer is too small. The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message. This pointer can also point to *null*. The function returns a negative integer in case of error, zero otherwise.

The `yapiGetDeviceInfo` function retrieves the information related to a module from its handle. `devdesc` is a 32-bit integer representing the module and which was obtained through `yapiGetAllDevices`. `infos` points to a data structure in which the result is stored. This data structure has the following format:

Name	Type	Size (bytes)	Description
vendorid	int	4	Yoctopuce USB ID
deviceid	int	4	Module USB ID
devrelease	int	4	Module version
nbinbterfaces	int	4	Number of USB interfaces used by the module
manufacturer	char[]	20	Yoctopuce (null terminated)
productname	char[]	28	Model (null terminated)
serial	char[]	20	Serial number (null terminated)
logicalname	char[]	20	Logical name (null terminated)
firmware	char[]	22	Firmware version (null terminated)
beacon	byte	1	Beacon state (0/1)

The `errmsg` parameter must point to a 255 character buffer to retrieve a potential error message.

Here is a sample program written in pascal using the `yapi.dll` DLL to list the connected modules.

```
// device description structure
type yDeviceSt = packed record
  vendorid      : word;
  deviceid      : word;
  devrelease    : word;
  nbinbterfaces : word;
  manufacturer  : array [0..19] of ansichar;
  productname   : array [0..27] of ansichar;
  serial        : array [0..19] of ansichar;
  logicalname    : array [0..19] of ansichar;
  firmware      : array [0..21] of ansichar;
  beacon        : byte;
end;

// Dll function import
function yapiInitAPI(mode:integer;
  errmsg : pansichar):integer;cdecl;
  external 'yapi.dll' name 'yapiInitAPI';

function yapiUpdateDeviceList(force:integer;errmsg : pansichar):integer;cdecl;
  external 'yapi.dll' name 'yapiUpdateDeviceList';

function yapiGetAllDevices( buffer:pointer;
  maxsize:integer;
  var neededsize:integer;
  errmsg : pansichar):integer; cdecl;
  external 'yapi.dll' name 'yapiGetAllDevices';

function apiGetDeviceInfo(d:integer; var infos:yDeviceSt;
  errmsg : pansichar):integer; cdecl;
  external 'yapi.dll' name 'yapiGetDeviceInfo';

var
  errmsgBuffer : array [0..256] of ansichar;
  dataBuffer    : array [0..127] of integer; // max of 128 USB devices
  errmsg,data   : pansichar;
  neededsize,i  : integer;
  devinfos      : yDeviceSt;

begin
  errmsg := @errmsgBuffer;

  // API initialization
  if(yapiInitAPI(1,errmsg)<0) then
  begin
    writeln(errmsg);
    halt;
  end;
```

```
// forces a device inventory
if( yapiUpdateDeviceList(1,errmsg)<0) then
begin
  writeln(errmsg);
  halt;
end;

// loads all device handles into dataBuffer
if yapiGetAllDevices(@dataBuffer,sizeof(dataBuffer),neededsize,errmsg)<0 then
begin
  writeln(errmsg);
  halt;
end;

// gets device info from each handle
for i:=0 to neededsize div sizeof(integer)-1 do
begin
  if (apiGetDeviceInfo(dataBuffer[i], devinfos, errmsg)<0) then
  begin
    writeln(errmsg);
    halt;
  end;
  writeln(pansichar(@devinfos.serial)+' ('+pansichar(@devinfos.productname)+')');
end;

end.
```

VB6 and yapi.dll

Each entry point from the yapi.dll is duplicated. You will find one regular C-decl version and one Visual Basic 6 compatible version, prefixed with *vb6_*.

21.5. Porting the high level library

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

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

22. Advanced programming

The preceding chapters have introduced, in each available language, the basic programming functions which can be used with your Yocto-Watt module. This chapter presents in a more generic manner a more advanced use of your module. Examples are provided in the language which is the most popular among Yoctopuce customers, that is C#. Nevertheless, you can find complete examples illustrating the concepts presented here in the programming libraries of each language.

To remain as concise as possible, examples provided in this chapter do not perform any error handling. Do not copy them "as is" in a production application.

22.1. Event programming

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

Detecting module arrival and departure

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

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

The callback

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

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

Initialization

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

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

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

Triggering callbacks

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

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

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

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

Detecting a modification in the value of a sensor

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

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

Callback invocation

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

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

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

The value change callback

This type of callback is called when a electrical power 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(YPower 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 YPower 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 Power function with the help of the `registerValueCallback` method. The following example sets up a callback for the first available Power function.

```
YPower f = YPower.FirstPower();
f.registerValueCallback(powerChangeCallBack)
```

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(YPower 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 Power 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 Power function.

```
YPower f = YPower.FirstPower();
f.set_reportFrequency("4/m");
```

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

```
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 `YPower`. Thus, the same callback function will be usable with any subclass of `YSensor` (and in particular with `YPower`). 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.

22.2. The data logger

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

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

Here are the main methods used to retrieve recorded measures:

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

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

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

// First call to loadMore() loads the summary/preview
dataset.loadMore();
YMeasure summary = dataset.get_summary();
string timeFmt = "dd MMM yyyy hh:mm:ss,fff";
string logFmt = "from {0} to {1} : average={2:0.00}{3}";
Console.WriteLine(String.Format(logFmt,
    summary.get_startTimeUTC_asDateTime().ToString(timeFmt),
    summary.get_endTimeUTC_asDateTime().ToString(timeFmt),
    summary.get_averageValue(), sensor.get_unit()));
```

² The `YMeasure` objects used by the data logger are exactly the same kind as those passed as argument to the timed report callbacks.

```
// Next calls to loadMore() will retrieve measures
Console.WriteLine("loading details");
int progress;
do {
    Console.Write(".");
    progress = dataset.loadMore();
} while(progress < 100);

// All measures have now been loaded
List<YMeasure> details = dataset.get_measures();
foreach (YMeasure m in details) {
    Console.WriteLine(String.Format(logFmt,
        m.get_startTimeUTC_asDateTime().ToString(timeFmt),
        m.get_endTimeUTC_asDateTime().ToString(timeFmt),
        m.get_averageValue(), sensor.get_unit()));
}
```

You will find a complete example demonstrating how to retrieve data from the logger for each programming language directly in the Yoctopuce library. The example can be found in directory *Examples/Prog-DataLogger*.

Timestamp

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

22.3. Sensor calibration

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

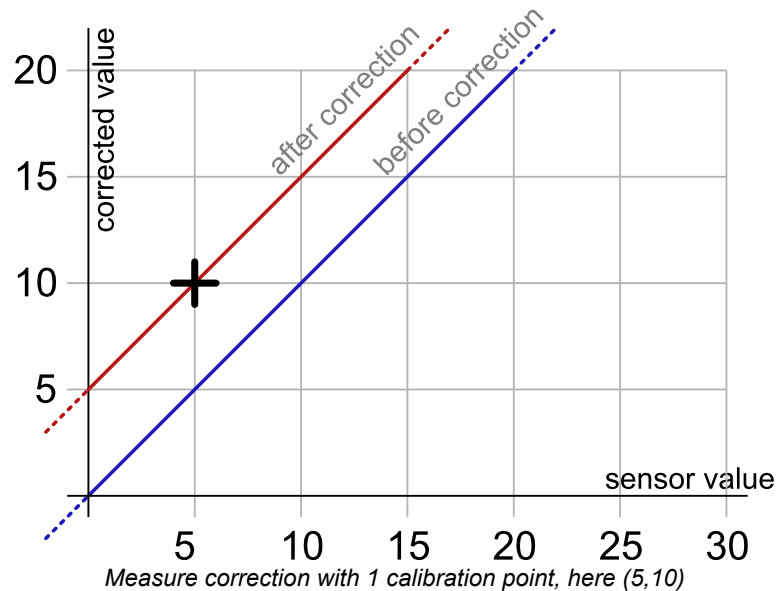
1 to 5 point linear interpolation

These transformations are performed directly inside the Yocto-Watt 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 power function returns *b*.

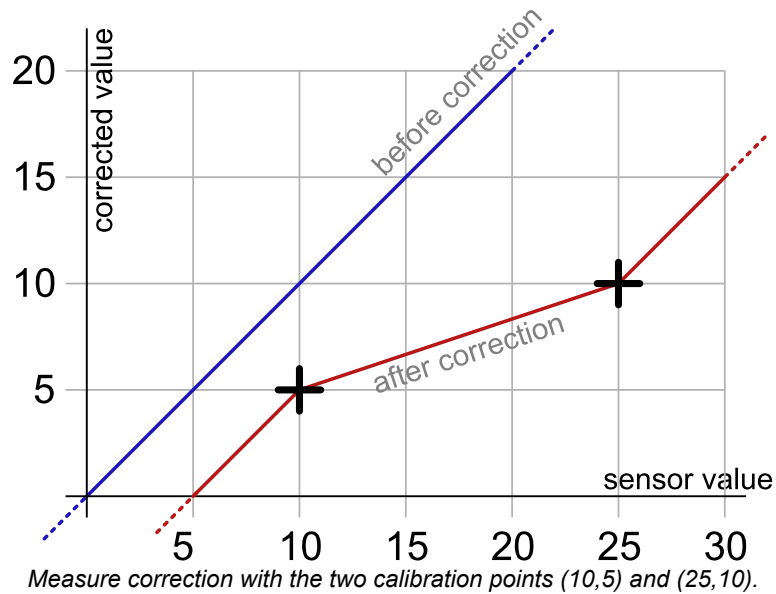


The application is very simple: you only need to call the `calibrateFromPoints()` method of the function you wish to correct. The following code applies the correction illustrated on the graph above to the first power function found. Note the call to the `saveToFlash` method of the module hosting the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

```
Double[] ValuesBefore = {5};
Double[] ValuesAfter  = {10};
YPower f = YPower.FirstPower();
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 power 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.



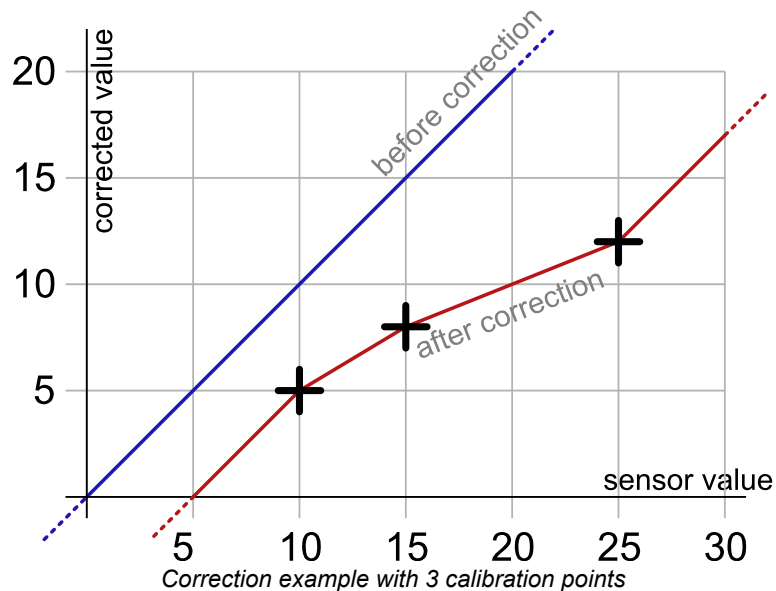
The code allowing you to program this calibration is very similar to the preceding code example.

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

```
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.

23. Firmware Update

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

23.1. The VirtualHub or the YoctoHub

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

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

23.2. The command line library

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

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

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

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

23.3. The Android application Yocto-Firmware

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

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

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

23.4. Updating the firmware with the programming library

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

Saving and restoring parameters

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

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

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

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

Finding the correct firmware

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

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

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

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

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

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

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

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

Updating the firmware

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

23.5. The "update" mode

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

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

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

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

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

```

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

```

24. High-level API Reference

This chapter summarizes the high-level API functions to drive your Yocto-Watt. Syntax and exact type names may vary from one language to another, but, unless otherwise stated, all the functions are available in every language. For detailed information regarding the types of arguments and return values for a given language, refer to the definition file for this language (`yocto_api.*` as well as the other `yocto_*` files that define the function interfaces).

For languages which support exceptions, all of these functions throw exceptions in case of error by default, rather than returning the documented error value for each function. This is by design, to facilitate debugging. It is however possible to disable the use of exceptions using the `yDisableExceptions()` function, in case you prefer to work with functions that return error values.

This chapter does not repeat the programming concepts described earlier, in order to stay as concise as possible. In case of doubt, do not hesitate to go back to the chapter describing in details all configurable attributes.

24.1. Class YAPI

General functions

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

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

java	<code>import com.yoctopuce.YoctoAPI.YAPI;</code>
dnf	<code>import YoctoProxyAPI.YAPIProxy</code>
cp	<code>#include "yocto_api_proxy.h"</code>
mL	<code>import YoctoProxyAPI.YAPIProxy"</code>
js	<code><script type='text/javascript' src='yocto_api.js'></script></code>
cpp	<code>#include "yocto_api.h"</code>
m	<code>#import "yocto_api.h"</code>
pas	<code>uses yocto_api;</code>
vb	<code>yocto_api.vb</code>
cs	<code>yocto_api.cs</code>
uwp	<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';</code> <code>in Node.js: import { YAPI, YErrorMsg, YModule, YSensor } from 'yoctolib-cjs/yocto_api_nodejs.js';</code>
es	<code>in HTML: <script src='../lib/yocto_api.js'></script></code> <code>in node.js: require('yoctolib-es2017/yocto_api.js');</code>
vi	<code>YModule.vi</code>

Global functions

YAPI.CheckLogicalName(name)

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

YAPI.ClearHTTPCallbackCacheDir(bool_removeFiles)

Disables the HTTP callback cache.

YAPI.DisableExceptions()

Disables the use of exceptions to report runtime errors.

YAPI.EnableExceptions()

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

YAPI.EnableUSBHost(osContext)

This function is used only on Android.

YAPI.FreeAPI()

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

YAPI.GetAPIVersion()

Returns the version identifier for the Yoctopuce library in use.

YAPI.GetCacheValidity()

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

YAPI.GetDeviceListValidity()

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

YAPI.GetDllArchitecture()

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

YAPI.GetDllPath()

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

YAPI.GetLog(lastLogLine)

Retrieves Yoctopuce low-level library diagnostic logs.

YAPI.GetNetworkTimeout()

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

YAPI.GetTickCount()

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

YAPI.HandleEvents(errmsg)

Maintains the device-to-library communication channel.

YAPI.InitAPI(mode, errmsg)

Initializes the Yoctopuce programming library explicitly.

YAPI.PreregisterHub(url, errmsg)

Fault-tolerant alternative to `yRegisterHub()`.

YAPI.RegisterDeviceArrivalCallback(arrivalCallback)

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

YAPI.RegisterDeviceRemovalCallback(removalCallback)

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

YAPI.RegisterHub(url, errmsg)

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

YAPI.RegisterHubDiscoveryCallback(hubDiscoveryCallback)

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

YAPI.RegisterHubWebsocketCallback(ws, errmsg, authpwd)

Variant to `yRegisterHub()` used to initialize Yoctopuce API on an existing Websocket session, as happens for incoming WebSocket callbacks.

YAPI.RegisterLogFunction(logfun)

Registers a log callback function.

YAPI.SelectArchitecture(arch)

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

YAPI.SetCacheValidity(cacheValidityMs)

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

YAPI.SetDelegate(object)

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

YAPI.SetDeviceListValidity(deviceListValidity)

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

YAPI.SetHTTPCallbackCacheDir(str_directory)

Enables the HTTP callback cache.

YAPI.SetNetworkTimeout(networkMsTimeout)

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

YAPI.SetTimeout(callback, ms_timeout, args)

Invoke the specified callback function after a given timeout.

YAPI.SetUSBPacketAckMs(pktAckDelay)

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

YAPI.Sleep(ms_duration, errmsg)

Pauses the execution flow for a specified duration.

YAPI.TestHub(url, mstimeout, errmsg)

Test if the hub is reachable.

YAPI.TriggerHubDiscovery(errmsg)

Force a hub discovery, if a callback as been registered with `yRegisterHubDiscoveryCallback` it will be called for each net work hub that will respond to the discovery.

YAPI.UnregisterHub(url)

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

YAPI.UpdateDeviceList(errmsg)

Triggers a (re)detection of connected Yoctopuce modules.

YAPI.UpdateDeviceList_async(callback, context)

Triggers a (re)detection of connected Yoctopuce modules.

YAPI.CheckLogicalName()**YAPI****YAPI.CheckLogicalName()**

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

js	function yCheckLogicalName (name)
cpp	bool CheckLogicalName (string name)
m	+(BOOL) CheckLogicalName :(NSString *) name
pas	boolean yCheckLogicalName (name : string): boolean
vb	function CheckLogicalName (ByVal name As String) As Boolean
cs	static bool CheckLogicalName (string name)
java	boolean CheckLogicalName (String name)
uwp	bool CheckLogicalName (string name)
py	CheckLogicalName (name)
php	function CheckLogicalName (\$name)
ts	async CheckLogicalName (name : string): Promise<boolean>
es	async CheckLogicalName (name)

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

Parameters :

name a string containing the name to check.

Returns :

true if the name is valid, false otherwise.

YAPI.ClearHTTPCallbackCacheDir() YAPI.ClearHTTPCallbackCacheDir()

YAPI

Disables the HTTP callback cache.

```
php function ClearHTTPCallbackCacheDir( $bool_removeFiles)
```

This method disables the HTTP callback cache, and can additionally cleanup the cache directory.

Parameters :

bool_removeFiles True to clear the content of the cache.

Returns :

nothing.

YAPI.DisableExceptions()

YAPI.DisableExceptions()

YAPI

Disables the use of exceptions to report runtime errors.

js	function yDisableExceptions ()
cpp	void DisableExceptions ()
m	+(void) DisableExceptions
pas	yDisableExceptions ()
vb	procedure DisableExceptions ()
cs	static void DisableExceptions ()
uwp	void DisableExceptions ()
py	DisableExceptions ()
php	function DisableExceptions ()
ts	async DisableExceptions (): Promise<void>
es	async DisableExceptions ()

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

YAPI.EnableExceptions()**YAPI****YAPI.EnableExceptions()**

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

js	function yEnableExceptions ()
cpp	void EnableExceptions ()
m	+(void) EnableExceptions
pas	yEnableExceptions ()
vb	procedure EnableExceptions ()
cs	static void EnableExceptions ()
uwp	void EnableExceptions ()
py	EnableExceptions ()
php	function EnableExceptions ()
ts	async EnableExceptions (): Promise<void>
es	async EnableExceptions ()

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

YAPI.EnableUSBHost()**YAPI****YAPI.EnableUSBHost()**

This function is used only on Android.

```
java void EnableUSBHost( Object osContext)
```

Before calling `yRegisterHub("usb")` you need to activate the USB host port of the system. This function takes as argument, an object of class `android.content.Context` (or any subclass). It is not necessary to call this function to reach modules through the network.

Parameters :

osContext an object of class `android.content.Context` (or any subclass).

YAPI.FreeAPI()**YAPI****YAPI.FreeAPI()**

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

js	function yFreeAPI ()
cpp	void FreeAPI ()
m	+(void) FreeAPI
pas	yFreeAPI ()
vb	procedure FreeAPI ()
cs	static void FreeAPI ()
java	void FreeAPI ()
uwp	void FreeAPI ()
py	FreeAPI ()
php	function FreeAPI ()
ts	async FreeAPI (): Promise<void>
es	async FreeAPI ()
dnp	static void FreeAPI ()
cp	static void FreeAPI ()

From an operating system standpoint, it is generally not required to call this function since the OS will automatically free allocated resources once your program is completed. However there are two situations when you may really want to use that function: - Free all dynamically allocated memory blocks in order to track a memory leak. - Send commands to devices right before the end of the program. Since commands are sent in an asynchronous way the program could exit before all commands are effectively sent. You should not call any other library function after calling **yFreeAPI()**, or your program will crash.

YAPI.GetAPIVersion() YAPI.GetAPIVersion()

YAPI

Returns the version identifier for the Yoctopuce library in use.

js	function yGetAPIVersion ()
cpp	string GetAPIVersion ()
m	+(NSString*) GetAPIVersion
pas	string yGetAPIVersion (): string
vb	function GetAPIVersion () As String
cs	static String GetAPIVersion ()
java	static String GetAPIVersion ()
uwp	static string GetAPIVersion ()
py	GetAPIVersion ()
php	function GetAPIVersion ()
ts	async GetAPIVersion ()
es	async GetAPIVersion ()
dnp	static string GetAPIVersion ()
cp	static string GetAPIVersion ()

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

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

Returns :

a character string describing the library version.

YAPI.GetCacheValidity()**YAPI****YAPI.GetCacheValidity()**

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

cpp	static u64 GetCacheValidity ()
m	+(u64) GetCacheValidity
pas	u64 yGetCacheValidity (): u64
vb	function GetCacheValidity () As Long
cs	ulong GetCacheValidity ()
java	long GetCacheValidity ()
uwp	async Task<ulong> GetCacheValidity ()
py	GetCacheValidity ()
php	function GetCacheValidity ()
ts	async GetCacheValidity (): Promise<number>
es	async GetCacheValidity ()

This method returns the cache validity of all attributes module functions. Note: This function must be called after `yInitAPI` .

Returns :

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

YAPI.GetDeviceListValidity()**YAPI****YAPI.GetDeviceListValidity()**

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

cpp	static int GetDeviceListValidity ()
m	+(int) GetDeviceListValidity
pas	LongInt yGetDeviceListValidity (): LongInt
vb	function GetDeviceListValidity () As Integer
cs	int GetDeviceListValidity ()
java	int GetDeviceListValidity ()
uwp	async Task<int> GetDeviceListValidity ()
py	GetDeviceListValidity ()
php	function GetDeviceListValidity ()
ts	async GetDeviceListValidity (): Promise<number>
es	async GetDeviceListValidity ()

Note: you must call this function after `yInitAPI`.

Returns :

the number of seconds between each enumeration.

YAPI.GetDllArchitecture()**YAPI****YAPI.GetDllArchitecture()**

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

```
static string GetDllArchitecture( )
```

On Windows, the architecture can be "Win32" or "Win64". On ARM machines, the architecture is "Armhf32" or "Aarch64". On other Linux machines, the architecture is "Linux32" or "Linux64". On MacOS, the architecture is "MacOs32" or "MacOs64".

Returns :

a character string describing the system architecture of the low-level communication library.

YAPI.GetDllPath()**YAPI****YAPI.GetDllPath()**

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

`static string GetDllPath( )`

For architectures that require multiple DLLs, in particular when using a .NET assembly DLL, the returned string takes the form "DotNetProxy=/...; yapi=/...;", where the first path corresponds to the .NET assembly DLL and the second path corresponds to the low-level communication library.

Returns :

a character string describing the DLL path.

YAPI.GetLog()**YAPI****YAPI.GetLog()**

Retrieves Yoctopuce low-level library diagnostic logs.

```
static string GetLog( string lastLogLine)
```

```
static string GetLog( string lastLogLine)
```

This method allows to progressively retrieve API logs. The interface is line-based: it must be called within a loop until the returned value is an empty string. Make sure to exit the loop when an empty string is returned, as feeding an empty string into the `lastLogLine` parameter for the next call would restart enumerating logs from the oldest message available.

Parameters :

lastLogLine On first call, provide an empty string. On subsequent calls, provide the last log line returned by `GetLog()`.

Returns :

a string with the log line immediately following the one given in argument, if such line exist. Returns an empty string otherwise, when completed.

YAPI.GetNetworkTimeout()

YAPI.GetNetworkTimeout()

YAPI

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

cpp	static int GetNetworkTimeout ()
m	+(int) GetNetworkTimeout
pas	LongInt yGetNetworkTimeout (): LongInt
vb	function GetNetworkTimeout () As Integer
cs	int GetNetworkTimeout ()
java	int GetNetworkTimeout ()
uwp	async Task<int> GetNetworkTimeout ()
py	GetNetworkTimeout ()
php	function GetNetworkTimeout ()
ts	async GetNetworkTimeout (): Promise<number>
es	async GetNetworkTimeout ()
dnp	static int GetNetworkTimeout ()
cp	static int GetNetworkTimeout ()

This delay impacts only the YoctoHubs and VirtualHub which are accessible through the network. By default, this delay is of 20000 milliseconds, but depending on your network you may want to change this delay, for example if your network infrastructure is based on a GSM connection.

Returns :

the network connection delay in milliseconds.

YAPI.GetTickCount()**YAPI****YAPI.GetTickCount()**

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

js	function yGetTickCount ()
cpp	u64 GetTickCount ()
m	+(u64) GetTickCount
pas	u64 yGetTickCount (): u64
vb	function GetTickCount () As Long
cs	static ulong GetTickCount ()
java	static long GetTickCount ()
uwp	static ulong GetTickCount ()
py	GetTickCount ()
php	function GetTickCount ()
ts	GetTickCount (): number
es	GetTickCount ()

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

Returns :

a long integer corresponding to the millisecond counter.

YAPI.HandleEvents() YAPI.HandleEvents()

YAPI

Maintains the device-to-library communication channel.

js	function yHandleEvents (errmsg)
cpp	YRETCODE HandleEvents (string errmsg)
m	+(YRETCODE) HandleEvents :(NSError**) errmsg
pas	integer yHandleEvents (var errmsg : string): integer
vb	function HandleEvents (ByRef errmsg As String) As YRETCODE
cs	static YRETCODE HandleEvents (ref string errmsg)
java	int HandleEvents ()
uwp	async Task<int> HandleEvents ()
py	HandleEvents (errmsg =None)
php	function HandleEvents (&\$ errmsg)
ts	async HandleEvents (errmsg : YErrorMsg null): Promise<number>
es	async HandleEvents (errmsg)

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

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

Parameters :

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

Returns :

YAPI . SUCCESS when the call succeeds.

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

YAPI.InitAPI()**YAPI.InitAPI()**

Initializes the Yoctopuce programming library explicitly.

js	function yInitAPI (mode , errmsg)
cpp	YRETCODE InitAPI (int mode , string errmsg)
m	+(YRETCODE) InitAPI :(int) mode :(NSError**) errmsg
pas	integer yInitAPI (mode : integer, var errmsg : string): integer
vb	function InitAPI (ByVal mode As Integer, ByRef errmsg As String) As Integer
cs	static int InitAPI (int mode , ref string errmsg)
java	int InitAPI (int mode)
uwp	async Task<int> InitAPI (int mode)
py	InitAPI (mode , errmsg =None)
php	function InitAPI (\$mode , & \$errmsg)
ts	async InitAPI (mode : number, errmsg : YErrorMsg): Promise<number>
es	async InitAPI (mode , errmsg)

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

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

Parameters :

mode an integer corresponding to the type of automatic device detection to use. Possible values are `YAPI.DetectNone`, `YAPI.DetectUSB`, `YAPI.DetectNet`, and `YAPI.DetectAll`.

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

Returns :

`YAPI.SUCCESS` when the call succeeds.

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

YAPI.PreregisterHub() YAPI.PreregisterHub()

YAPI

Fault-tolerant alternative to `yRegisterHub()`.

js	<code>function yPreregisterHub(url, errmsg)</code>
cpp	<code>YRETCODE PreregisterHub(string url, string errmsg)</code>
m	<code>+(YRETCODE) PreregisterHub :(NSString *) url :(NSError**) errmsg</code>
pas	<code>integer yPreregisterHub(url: string, var errmsg: string): integer</code>
vb	<code>function PreregisterHub(ByVal url As String, ByRef errmsg As String) As Integer</code>
cs	<code>static int PreregisterHub(string url, ref string errmsg)</code>
java	<code>int PreregisterHub(String url)</code>
uwp	<code>async Task<int> PreregisterHub(string url)</code>
py	<code>PreregisterHub(url, errmsg=None)</code>
php	<code>function PreregisterHub(\$url, &\$errmsg)</code>
ts	<code>async PreregisterHub(url: string, errmsg: YErrorMsg): Promise<number></code>
es	<code>async PreregisterHub(url, errmsg)</code>
dnp	<code>static string PreregisterHub(string url)</code>
cp	<code>static string PreregisterHub(string url)</code>

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

Parameters :

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

Returns :

YAPI . SUCCESS when the call succeeds.

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

YAPI.RegisterDeviceArrivalCallback()**YAPI****YAPI.RegisterDeviceArrivalCallback()**

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

js	function yRegisterDeviceArrivalCallback (arrivalCallback)
cpp	void RegisterDeviceArrivalCallback (yDeviceUpdateCallback arrivalCallback)
m	+(void) RegisterDeviceArrivalCallback :(yDeviceUpdateCallback) arrivalCallback
pas	yRegisterDeviceArrivalCallback (arrivalCallback : yDeviceUpdateFunc)
vb	procedure RegisterDeviceArrivalCallback (ByVal arrivalCallback As yDeviceUpdateFunc)
cs	static void RegisterDeviceArrivalCallback (yDeviceUpdateFunc arrivalCallback)
java	void RegisterDeviceArrivalCallback (DeviceArrivalCallback arrivalCallback)
uwp	void RegisterDeviceArrivalCallback (DeviceUpdateHandler arrivalCallback)
py	RegisterDeviceArrivalCallback (arrivalCallback)
php	function RegisterDeviceArrivalCallback (\$arrivalCallback)
ts	async RegisterDeviceArrivalCallback (arrivalCallback : YDeviceUpdateCallback null): Promise<void>
es	async RegisterDeviceArrivalCallback (arrivalCallback)

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

Parameters :

arrivalCallback a procedure taking a `YModule` parameter, or `null`

YAPI.RegisterDeviceRemovalCallback()

YAPI.RegisterDeviceRemovalCallback()

YAPI

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

js	function yRegisterDeviceRemovalCallback (removalCallback)
cpp	void RegisterDeviceRemovalCallback (yDeviceUpdateCallback removalCallback)
m	+(void) RegisterDeviceRemovalCallback :(yDeviceUpdateCallback) removalCallback
pas	yRegisterDeviceRemovalCallback (removalCallback : yDeviceUpdateFunc)
vb	procedure RegisterDeviceRemovalCallback (ByVal removalCallback As yDeviceUpdateFunc)
cs	static void RegisterDeviceRemovalCallback (yDeviceUpdateFunc removalCallback)
java	void RegisterDeviceRemovalCallback (DeviceRemovalCallback removalCallback)
uwp	void RegisterDeviceRemovalCallback (DeviceUpdateHandler removalCallback)
py	RegisterDeviceRemovalCallback (removalCallback)
php	function RegisterDeviceRemovalCallback (\$removalCallback)
ts	async RegisterDeviceRemovalCallback (removalCallback : YDeviceUpdateCallback null): Promise<void>
es	async RegisterDeviceRemovalCallback (removalCallback)

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

Parameters :

removalCallback a procedure taking a YModule parameter, or null

YAPI.RegisterHub()**YAPI.RegisterHub()**

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

js	function yRegisterHub (url, errmsg)
cpp	YRETCODE RegisterHub (string url, string errmsg)
m	+(YRETCODE) RegisterHub :(NSString *) url :(NSError**) errmsg
pas	integer yRegisterHub (url: string, var errmsg: string): integer
vb	function RegisterHub (ByVal url As String, ByRef errmsg As String) As Integer
cs	static int RegisterHub (string url, ref string errmsg)
java	int RegisterHub (String url)
uwp	async Task<int> RegisterHub (string url)
py	RegisterHub (url, errmsg=None)
php	function RegisterHub (\$url, &\$errmsg)
ts	async RegisterHub (url: string, errmsg: YErrorMsg): Promise<number>
es	async RegisterHub (url, errmsg)
dnp	static string RegisterHub (string url)
cp	static string RegisterHub (string url)

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

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

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

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

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

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

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

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

Parameters :

url a string containing either "usb", "callback" or the root URL of the hub to monitor

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

Returns :

YAPI . SUCCESS when the call succeeds.

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

YAPI.RegisterHubDiscoveryCallback()**YAPI****YAPI.RegisterHubDiscoveryCallback()**

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

cpp	void RegisterHubDiscoveryCallback (YHubDiscoveryCallback hubDiscoveryCallback)
m	+(void) RegisterHubDiscoveryCallback : (YHubDiscoveryCallback) hubDiscoveryCallback
pas	yRegisterHubDiscoveryCallback (hubDiscoveryCallback : YHubDiscoveryCallback)
vb	procedure RegisterHubDiscoveryCallback (ByVal hubDiscoveryCallback As YHubDiscoveryCallback)
cs	static void RegisterHubDiscoveryCallback (YHubDiscoveryCallback hubDiscoveryCallback)
java	void RegisterHubDiscoveryCallback (HubDiscoveryCallback hubDiscoveryCallback)
uwp	async Task RegisterHubDiscoveryCallback (HubDiscoveryHandler hubDiscoveryCallback)
py	RegisterHubDiscoveryCallback (hubDiscoveryCallback)
ts	async RegisterHubDiscoveryCallback (hubDiscoveryCallback : YHubDiscoveryCallback): Promise<number>
es	async RegisterHubDiscoveryCallback (hubDiscoveryCallback)

The callback has two string parameter, the first one contain the serial number of the hub and the second contain the URL of the network hub (this URL can be passed to RegisterHub). This callback will be invoked while yUpdateDeviceList is running. You will have to call this function on a regular basis.

Parameters :

hubDiscoveryCallback a procedure taking two string parameter, the serial

YAPI.RegisterHubWebsocketCallback()**YAPI****YAPI.RegisterHubWebsocketCallback()**

Variant to `yRegisterHub()` used to initialize Yoctopuce API on an existing Websocket session, as happens for incoming WebSocket callbacks.

Parameters :

- ws** node WebSocket object for the incoming WebSocket callback connection
- errmsg** a string passed by reference to receive any error message.
- authpwd** the optional authentication password, required only authentication is configured on the calling hub.

Returns :

`YAPI.SUCCESS` when the call succeeds.

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

YAPI.RegisterLogFunction() YAPI.RegisterLogFunction()

YAPI

Registers a log callback function.

cpp	void RegisterLogFunction (yLogFunction logfun)
m	+(void) RegisterLogFunction :(yLogCallback) logfun
pas	yRegisterLogFunction (logfun : yLogFunc)
vb	procedure RegisterLogFunction (ByVal logfun As yLogFunc)
cs	static void RegisterLogFunction (yLogFunc logfun)
java	void RegisterLogFunction (LogCallback logfun)
uwp	void RegisterLogFunction (LogHandler logfun)
py	RegisterLogFunction (logfun)
ts	async RegisterLogFunction (logfun : YLogCallback): Promise<number>
es	async RegisterLogFunction (logfun)

This callback will be called each time the API have something to say. Quite useful to debug the API.

Parameters :

logfun a procedure taking a string parameter, or null

YAPI.SelectArchitecture() YAPI.SelectArchitecture()

YAPI

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

```
py SelectArchitecture( arch)
```

By default, the Python library automatically detects the appropriate library to use. However, for Linux ARM, it not possible to reliably distinguish between a Hard Float (armhf) and a Soft Float (armel) install. For in this case, it is therefore recommended to manually select the proper architecture by calling `SelectArchitecture( )` before any other call to the library.

Parameters :

arch A string containing the architecture to use. Possibles value are: "armhf","armel", "aarch64","i386","x86_64", "32bit", "64bit"

Returns :

nothing.

On failure, throws an exception.

YAPI.SetCacheValidity()**YAPI****YAPI.SetCacheValidity()**

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

cpp	static void SetCacheValidity (u64 cacheValidityMs)
m	+(void) SetCacheValidity : (u64) cacheValidityMs
pas	ySetCacheValidity (cacheValidityMs : u64)
vb	procedure SetCacheValidity (ByVal cacheValidityMs As Long)
cs	void SetCacheValidity (ulong cacheValidityMs)
java	void SetCacheValidity (long cacheValidityMs)
uwp	async Task SetCacheValidity (ulong cacheValidityMs)
py	SetCacheValidity (cacheValidityMs)
php	function SetCacheValidity (\$ cacheValidityMs)
ts	async SetCacheValidity (cacheValidityMs : number): Promise<void>
es	async SetCacheValidity (cacheValidityMs)

By default, when accessing a module, all the attributes of the module functions are automatically kept in cache for the standard duration (5 ms). This method can be used to change this standard duration, for example in order to reduce network or USB traffic. This parameter does not affect value change callbacks Note: This function must be called after `yInitAPI`.

Parameters :

cacheValidityMs an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds.

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

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

m

+(void) SetDelegate : (id) object

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

Parameters :

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

YAPI.SetDeviceListValidity()**YAPI****YAPI.SetDeviceListValidity()**

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

cpp	static void SetDeviceListValidity (int deviceListValidity)
m	+(void) SetDeviceListValidity : (int) deviceListValidity
pas	ySetDeviceListValidity (deviceListValidity : LongInt)
vb	procedure SetDeviceListValidity (ByVal deviceListValidity As Integer)
cs	void SetDeviceListValidity (int deviceListValidity)
java	void SetDeviceListValidity (int deviceListValidity)
uwp	async Task SetDeviceListValidity (int deviceListValidity)
py	SetDeviceListValidity (deviceListValidity)
php	function SetDeviceListValidity (\$ deviceListValidity)
ts	async SetDeviceListValidity (deviceListValidity : number): Promise<void>
es	async SetDeviceListValidity (deviceListValidity)

By default, the library performs a full enumeration every 10 seconds. To reduce network traffic, you can increase this delay. It's particularly useful when a YoctoHub is connected to the GSM network where traffic is billed. This parameter doesn't impact modules connected by USB, nor the working of module arrival/removal callbacks. Note: you must call this function after `yInitAPI`.

Parameters :

deviceListValidity number of seconds between each enumeration.

YAPI.SetHTTPCallbackCacheDir() YAPI.SetHTTPCallbackCacheDir()

YAPI

Enables the HTTP callback cache.

```
php function SetHTTPCallbackCacheDir( $str_directory)
```

When enabled, this cache reduces the quantity of data sent to the PHP script by 50% to 70%. To enable this cache, the method `ySetHTTPCallbackCacheDir()` must be called before any call to `yRegisterHub()`. This method takes in parameter the path of the directory used for saving data between each callback. This folder must exist and the PHP script needs to have write access to it. It is recommended to use a folder that is not published on the Web server since the library will save some data of Yoctopuce devices into this folder.

Note: This feature is supported by YoctoHub and VirtualHub since version 27750.

Parameters :

str_directory the path of the folder that will be used as cache.

Returns :

nothing.

On failure, throws an exception.

YAPI.SetNetworkTimeout()

YAPI.SetNetworkTimeout()

YAPI

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

cpp	<code>static void SetNetworkTimeout(int networkMsTimeout)</code>
m	<code>+(void) SetNetworkTimeout : (int) networkMsTimeout</code>
pas	<code>ySetNetworkTimeout(networkMsTimeout: LongInt)</code>
vb	<code>procedure SetNetworkTimeout(ByVal networkMsTimeout As Integer)</code>
cs	<code>void SetNetworkTimeout(int networkMsTimeout)</code>
java	<code>void SetNetworkTimeout(int networkMsTimeout)</code>
uwp	<code>async Task SetNetworkTimeout(int networkMsTimeout)</code>
py	<code>SetNetworkTimeout(networkMsTimeout)</code>
php	<code>function SetNetworkTimeout(\$networkMsTimeout)</code>
ts	<code>async SetNetworkTimeout(networkMsTimeout: number): Promise<void></code>
es	<code>async SetNetworkTimeout(networkMsTimeout)</code>
dnp	<code>static void SetNetworkTimeout(int networkMsTimeout)</code>
cp	<code>static void SetNetworkTimeout(int networkMsTimeout)</code>

This delay impacts only the YoctoHubs and VirtualHub which are accessible through the network. By default, this delay is of 20000 milliseconds, but depending on your network you may want to change this delay, for example if your network infrastructure is based on a GSM connection.

Parameters :

networkMsTimeout the network connection delay in milliseconds.

YAPI.SetTimeout()**YAPI****YAPI.SetTimeout()**

Invoke the specified callback function after a given timeout.

js	function ySetTimeout (callback , ms_timeout , args)
ts	SetTimeout (callback : Function, ms_timeout : number): number
es	SetTimeout (callback , ms_timeout , args)

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

Parameters :

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

Returns :

YAPI . SUCCESS when the call succeeds.

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

YAPI.SetUSBPacketAckMs()**YAPI****YAPI.SetUSBPacketAckMs()**

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

```
java void SetUSBPacketAckMs( int pktAckDelay)
```

This function allows the library to run on Android phones that tend to loose USB packets. By default, this feature is disabled because it doubles the number of packets sent and slows down the API considerably. Therefore, the acknowledge of incoming USB packets should only be enabled on phones or tablets that loose USB packets. A delay of 50 milliseconds is generally enough. In case of doubt, contact Yoctopuce support. To disable USB packets acknowledge, call this function with the value 0. Note: this feature is only available on Android.

Parameters :

pktAckDelay then number of milliseconds before the module

YAPI.Sleep()**YAPI****YAPI.Sleep()**

Pauses the execution flow for a specified duration.

js	function ySleep (ms_duration , errmsg)
cpp	YRETCODE Sleep (unsigned ms_duration , string errmsg)
m	+(YRETCODE) Sleep :(unsigned) ms_duration :(NSError **) errmsg
pas	integer ySleep (ms_duration : integer, var errmsg : string): integer
vb	function Sleep (ByVal ms_duration As Integer, ByRef errmsg As String) As Integer
cs	static int Sleep (int ms_duration , ref string errmsg)
java	int Sleep (long ms_duration)
uwp	async Task<int> Sleep (ulong ms_duration)
py	Sleep (ms_duration , errmsg =None)
php	function Sleep (\$ms_duration , &\$errmsg)
ts	async Sleep (ms_duration : number, errmsg : YErrorMsg null): Promise<number>
es	async Sleep (ms_duration , errmsg)

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

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

Parameters :

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

Returns :

YAPI.SUCCESS when the call succeeds.

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

YAPI.TestHub()**YAPI.TestHub()**

Test if the hub is reachable.

```

cpp YRETCODE TestHub( string url, int mtimeout, string errmsg)
m   +(YRETCODE) TestHub : (NSString*) url
                                : (int) mtimeout
                                : (NSError**) errmsg
pas integer yTestHub( url: string,
                      mtimeout: integer,
                      var errmsg: string): integer
vb   function TestHub( ByVal url As String,
                      ByVal mtimeout As Integer,
                      ByRef errmsg As String) As Integer
cs   static int TestHub( string url, int mtimeout, ref string errmsg)
java int TestHub( String url, int mtimeout)
uwp  async Task<int> TestHub( string url, uint mtimeout)
py   TestHub( url, mtimeout, errmsg=None)
php  function TestHub( $url, $mtimeout, &$errmsg)
ts   async TestHub( url: string, mtimeout: number, errmsg: YErrorMsg): Promise<number>
es   async TestHub( url, mtimeout, errmsg)
dnp  static string TestHub( string url, int mtimeout)
cp   static string TestHub( string url, int mtimeout)

```

This method do not register the hub, it only test if the hub is usable. The url parameter follow the same convention as the yRegisterHub method. This method is useful to verify the authentication parameters for a hub. It is possible to force this method to return after mtimeout milliseconds.

Parameters :

- url** a string containing either "usb", "callback" or the root URL of the hub to monitor
- mtimeout** the number of millisecond available to test the connection.
- errmsg** a string passed by reference to receive any error message.

Returns :

YAPI . SUCCESS when the call succeeds.

On failure returns a negative error code.

YAPI.TriggerHubDiscovery()**YAPI****YAPI.TriggerHubDiscovery()**

Force a hub discovery, if a callback as been registered with `yRegisterHubDiscoveryCallback` it will be called for each net work hub that will respond to the discovery.

cpp	<code>YRETCODE TriggerHubDiscovery(string errmsg)</code>
m	<code>+(YRETCODE) TriggerHubDiscovery : (NSError**) errmsg</code>
pas	<code>integer yTriggerHubDiscovery(var errmsg: string): integer</code>
vb	<code>function TriggerHubDiscovery(ByRef errmsg As String) As Integer</code>
cs	<code>static int TriggerHubDiscovery(ref string errmsg)</code>
java	<code>int TriggerHubDiscovery()</code>
uwp	<code>Task<int> TriggerHubDiscovery()</code>
py	<code>TriggerHubDiscovery(errmsg=None)</code>
ts	<code>async TriggerHubDiscovery(errmsg: YErrorMsg null): Promise<number></code>
es	<code>async TriggerHubDiscovery(errmsg)</code>

Parameters :

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

Returns :

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

YAPI.UnregisterHub()**YAPI****YAPI.UnregisterHub()**

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

js	function yUnregisterHub (url)
cpp	void UnregisterHub (string url)
m	+(void) UnregisterHub :(NSString *) url
pas	yUnregisterHub (url : string)
vb	procedure UnregisterHub (ByVal url As String)
cs	static void UnregisterHub (string url)
java	void UnregisterHub (String url)
uwp	async Task UnregisterHub (string url)
py	UnregisterHub (url)
php	function UnregisterHub (\$url)
ts	async UnregisterHub (url : string): Promise<void>
es	async UnregisterHub (url)

Parameters :

url a string containing either "usb" or the

YAPI.UpdateDeviceList()**YAPI****YAPI.UpdateDeviceList()**

Triggers a (re)detection of connected Yoctopuce modules.

js	<code>function yUpdateDeviceList(errmsg)</code>
cpp	<code>YRETCODE UpdateDeviceList(string errmsg)</code>
m	<code>+(YRETCODE) UpdateDeviceList :(NSError**) errmsg</code>
pas	<code>integer yUpdateDeviceList(var errmsg: string): integer</code>
vb	<code>function UpdateDeviceList(ByRef errmsg As String) As YRETCODE</code>
cs	<code>static YRETCODE UpdateDeviceList(ref string errmsg)</code>
java	<code>int UpdateDeviceList()</code>
uwp	<code>async Task<int> UpdateDeviceList()</code>
py	<code>UpdateDeviceList(errmsg=None)</code>
php	<code>function UpdateDeviceList(&\$errmsg)</code>
ts	<code>async UpdateDeviceList(errmsg: YErrorMsg null): Promise<number></code>
es	<code>async UpdateDeviceList(errmsg)</code>

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

This function can be called as frequently as desired to refresh the device list and to make the application aware of hot-plug events. However, since device detection is quite a heavy process, `UpdateDeviceList` shouldn't be called more than once every two seconds.

Parameters :

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

Returns :

`YAPI . SUCCESS` when the call succeeds.

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

YAPI.UpdateDeviceList_async()**YAPI****YAPI.UpdateDeviceList_async()**

Triggers a (re)detection of connected Yoctopuce modules.

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

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

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

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

Parameters :

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

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

Returns :

nothing : the result is provided to the callback.

24.2. Class YModule

Global parameters control interface for all Yoctopuce devices

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

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

js	<script type='text/javascript' src='yocto_api.js'></script>
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
uwp	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *
php	require_once('yocto_api.php');
ts	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';
es	in HTML: <script src="../lib/yocto_api.js"></script> in node.js: require('yoctolib-es2017/yocto_api.js');
dnp	import YoctoProxyAPI.YModuleProxy
cp	#include "yocto_module_proxy.h"
vi	YModule.vi
ml	import YoctoProxyAPI.YModuleProxy

Global functions

YModule.FindModule(func)

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

YModule.FindModuleInContext(yctx, func)

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

YModule.FirstModule()

Starts the enumeration of modules currently accessible.

YModule properties

module→Beacon [writable]

State of the localization beacon.

module→FirmwareRelease [read-only]

Version of the firmware embedded in the module.

module→FunctionId [read-only]

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

module→HardwareId [read-only]

Unique hardware identifier of the module.

module→IsOnline [read-only]

Checks if the module is currently reachable.

module→LogicalName [writable]

Logical name of the module.

module→Luminosity [writable]

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

module→**ProductId** [*read-only*]

USB device identifier of the module.

module→**ProductName** [*read-only*]

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

module→**ProductRelease** [*read-only*]

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

module→**SerialNumber** [*read-only*]

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

YModule methods

module→**checkFirmware**(**path**, **onlynew**)

Tests whether the byn file is valid for this module.

module→**clearCache**()

Invalidates the cache.

module→**describe**()

Returns a descriptive text that identifies the module.

module→**download**(**pathname**)

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

module→**functionBaseType**(**functionIndex**)

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

module→**functionCount**()

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

module→**functionId**(**functionIndex**)

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

module→**functionName**(**functionIndex**)

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

module→**functionType**(**functionIndex**)

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

module→**functionValue**(**functionIndex**)

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

module→**get_allSettings**()

Returns all the settings and uploaded files of the module.

module→**get_beacon**()

Returns the state of the localization beacon.

module→**get_errorMessage**()

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

module→**get_errorType**()

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

module→**get_firmwareRelease**()

Returns the version of the firmware embedded in the module.

module→**get_functionIds**(**funType**)

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

module→**get_hardwareId**()

Returns the unique hardware identifier of the module.

module→**get_icon2d**()

Returns the icon of the module.
module→get_lastLogs() Returns a string with last logs of the module.
module→get_logicalName() Returns the logical name of the module.
module→get_luminosity() Returns the luminosity of the module informative LEDs (from 0 to 100).
module→get_parentHub() Returns the serial number of the YoctoHub on which this module is connected.
module→get_persistentSettings() Returns the current state of persistent module settings.
module→get_productId() Returns the USB device identifier of the module.
module→get_productName() Returns the commercial name of the module, as set by the factory.
module→get_productRelease() Returns the release number of the module hardware, preprogrammed at the factory.
module→get_rebootCountdown() Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.
module→get_serialNumber() Returns the serial number of the module, as set by the factory.
module→get_subDevices() Returns a list of all the modules that are plugged into the current module.
module→get_upTime() Returns the number of milliseconds spent since the module was powered on.
module→get_url() Returns the URL used to access the module.
module→get_usbCurrent() Returns the current consumed by the module on the USB bus, in milli-amps.
module→get_userData() Returns the value of the userData attribute, as previously stored using method <code>set_userData</code> .
module→get_userVar() Returns the value previously stored in this attribute.
module→hasFunction(funcId) Tests if the device includes a specific function.
module→isOnline() Checks if the module is currently reachable, without raising any error.
module→isOnline_async(callback, context) Checks if the module is currently reachable, without raising any error.
module→load(msValidity) Preloads the module cache with a specified validity duration.
module→load_async(msValidity, callback, context) Preloads the module cache with a specified validity duration (asynchronous version).
module→log(text) Adds a text message to the device logs.

module→nextModule()

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

module→reboot(secBeforeReboot)

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

module→registerBeaconCallback(callback)

Register a callback function, to be called when the localization beacon of the module has been changed.

module→registerConfigChangeCallback(callback)

Register a callback function, to be called when a persistent settings in a device configuration has been changed (e.g.

module→registerLogCallback(callback)

Registers a device log callback function.

module→revertFromFlash()

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

module→saveToFlash()

Saves current settings in the nonvolatile memory of the module.

module→set_allSettings(settings)

Restores all the settings of the device.

module→set_allSettingsAndFiles(settings)

Restores all the settings and uploaded files to the module.

module→set_beacon(newval)

Turns on or off the module localization beacon.

module→set_logicalName(newval)

Changes the logical name of the module.

module→set_luminosity(newval)

Changes the luminosity of the module informative leds.

module→set_userData(data)

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

module→set_userVar(newval)

Stores a 32 bit value in the device RAM.

module→triggerConfigChangeCallback()

Triggers a configuration change callback, to check if they are supported or not.

module→triggerFirmwareUpdate(secBeforeReboot)

Schedules a module reboot into special firmware update mode.

module→updateFirmware(path)

Prepares a firmware update of the module.

module→updateFirmwareEx(path, force)

Prepares a firmware update of the module.

module→wait_async(callback, context)

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

YModule.FindModule()**YModule****YModule.FindModule()**

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

js	function yFindModule (func)
cpp	YModule* FindModule (string func)
m	+(YModule*) FindModule : (NSString*) func
pas	TYModule yFindModule (func : string): TYModule
vb	function FindModule (ByVal func As String) As YModule
cs	static YModule FindModule (string func)
java	static YModule FindModule (String func)
uwp	static YModule FindModule (string func)
py	FindModule (func)
php	function FindModule (\$func)
ts	static FindModule (func : string): YModule
es	static FindModule (func)
dnp	static YModuleProxy FindModule (string func)
cp	static YModuleProxy * FindModule (string func)

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

If a call to this object's `is_online()` method returns FALSE although you are certain that the device is plugged, make sure that you did call `registerHub()` at application initialization time.

Parameters :

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

Returns :

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

YModule.FindModuleInContext()**YModule****YModule.FindModuleInContext()**

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

java	static YModule FindModuleInContext (YAPIContext yctx , String func)
uwp	static YModule FindModuleInContext (YAPIContext yctx , string func)
ts	static FindModuleInContext (yctx : YAPIContext, func : string): YModule
es	static FindModuleInContext (yctx , func)

The identifier can be specified using several formats:

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

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

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the module, for instance `MyDevice.module`.

Returns :

a `YModule` object allowing you to drive the module.

YModule.FirstModule()**YModule****YModule.FirstModule()**

Starts the enumeration of modules currently accessible.

js	function yFirstModule ()
cpp	YModule * FirstModule ()
m	+(YModule*) FirstModule
pas	TYModule yFirstModule (): TYModule
vb	function FirstModule () As YModule
cs	static YModule FirstModule ()
java	static YModule FirstModule ()
uwp	static YModule FirstModule ()
py	FirstModule ()
php	function FirstModule ()
ts	static FirstModule (): YModule null
es	static FirstModule ()

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

Returns :

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

module→**Beacon****YModule**

State of the localization beacon.

dnf

int Beacon

Writable. Turns on or off the module localization beacon.

module→FirmwareRelease**YModule**

Version of the firmware embedded in the module.

dnp

[string FirmwareRelease](#)

module→FunctionId**YModule**

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

dnp

 string **FunctionId**

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

module→**HardwareId****YModule**

Unique hardware identifier of the module.

dnf [string **HardwareId**](#)

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

module→IsOnline**YModule**

Checks if the module is currently reachable.

dnp

bool IsOnline

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

module→LogicalName**YModule**

Logical name of the module.

dnf `string LogicalName`

Writable. You can use `yCheckLogicalName( )` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

module→Luminosity**YModule**

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

dnp

[int Luminosity](#)

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

module→ProductId**YModule**

USB device identifier of the module.

dnp	int ProductId

module→ProductName**YModule**

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

dnp

 string **ProductName**

module→ProductRelease**YModule**

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

dnp `int` **ProductRelease**

The original hardware release returns value 1, revision B returns value 2, etc.

module→**SerialNumber****YModule**

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

dnf

 string **SerialNumber**

module→checkFirmware()**YModule**

Tests whether the byn file is valid for this module.

js	function checkFirmware (path , onlynew)
cpp	string checkFirmware (string path , bool onlynew)
m	-(NSString*) checkFirmware : (NSString*) path : (bool) onlynew
pas	string checkFirmware (path : string, onlynew : boolean): string
vb	function checkFirmware (ByVal path As String, ByVal onlynew As Boolean) As String
cs	string checkFirmware (string path , bool onlynew)
java	String checkFirmware (String path , boolean onlynew)
uwp	async Task<string> checkFirmware (string path , bool onlynew)
py	checkFirmware (path , onlynew)
php	function checkFirmware (\$ path , \$ onlynew)
ts	async checkFirmware (path : string, onlynew : boolean): Promise<string>
es	async checkFirmware (path , onlynew)
dnp	string checkFirmware (string path , bool onlynew)
cp	string checkFirmware (string path , bool onlynew)
cmd	YModule target checkFirmware path onlynew

This method is useful to test if the module needs to be updated. It is possible to pass a directory as argument instead of a file. In this case, this method returns the path of the most recent appropriate .byn file. If the parameter onLynew is true, the function discards firmwares that are older or equal to the installed firmware.

Parameters :

- path** the path of a byn file or a directory that contains byn files
- onlynew** returns only files that are strictly newer

Returns :

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

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

module→clearCache()**YModule**

Invalidates the cache.

js	function clearCache ()
cpp	void clearCache ()
m	-(void) clearCache
pas	clearCache ()
vb	procedure clearCache ()
cs	void clearCache ()
java	void clearCache ()
py	clearCache ()
php	function clearCache ()
ts	async clearCache (): Promise<void>
es	async clearCache ()

Invalidates the cache of the module attributes. Forces the next call to `get_xxx()` or `loadxxx()` to use values that come from the device.

module→describe()**YModule**

Returns a descriptive text that identifies the module.

js	function describe ()
cpp	string describe ()
m	-(NSString*) describe
pas	string describe (): string
vb	function describe () As String
cs	string describe ()
java	String describe ()
py	describe ()
php	function describe ()
ts	async describe (): Promise<string>
es	async describe ()

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

Returns :

a string that describes the module

module→download()**YModule**

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

js	function download (pathname)
cpp	string download (string pathname)
m	-(NSMutableData*) download : (NSString*) pathname
pas	TByteArray download (pathname : string): TByteArray
vb	function download (ByVal pathname As String) As Byte
cs	byte[] download (string pathname)
java	byte[] download (String pathname)
uwp	async Task<byte[]> download (string pathname)
py	download (pathname)
php	function download (\$pathname)
ts	async download (pathname : string): Promise<Uint8Array>
es	async download (pathname)
dnp	byte[] download (string pathname)
cp	string download (string pathname)
cmd	YModule target download pathname

Parameters :

pathname name of the new file to load

Returns :

a binary buffer with the file content

On failure, throws an exception or returns YAPI_INVALID_STRING.

module→**functionBaseType()****YModule**

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

js	function functionBaseType (functionIndex)
cpp	string functionBaseType (int functionIndex)
pas	string functionBaseType (functionIndex : integer): string
vb	function functionBaseType (ByVal functionIndex As Integer) As String
cs	string functionBaseType (int functionIndex)
java	String functionBaseType (int functionIndex)
py	functionBaseType (functionIndex)
php	function functionBaseType (\$ functionIndex)
ts	async functionBaseType (functionIndex : number): Promise<string>
es	async functionBaseType (functionIndex)

For instance, the base type of all measuring functions is "Sensor".

Parameters :

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

Returns :

a string corresponding to the base type of the function

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

module→functionCount()**YModule**

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

js	function functionCount ()
cpp	int functionCount ()
m	-(int) functionCount
pas	integer functionCount (): integer
vb	function functionCount () As Integer
cs	int functionCount ()
java	int functionCount ()
py	functionCount ()
php	function functionCount ()
ts	async functionCount (): Promise<number>
es	async functionCount ()

Returns :

the number of functions on the module

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

module→functionId()**YModule**

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

js	function functionId (functionIndex)
cpp	string functionId (int functionIndex)
m	-(NSString*) functionId : (int) functionIndex
pas	string functionId (functionIndex : integer): string
vb	function functionId (ByVal functionIndex As Integer) As String
cs	string functionId (int functionIndex)
java	String functionId (int functionIndex)
py	functionId (functionIndex)
php	function functionId (\$ functionIndex)
ts	async functionId (functionIndex : number): Promise<string>
es	async functionId (functionIndex)

Parameters :

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

Returns :

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

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

module→functionName()**YModule**

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

js	function functionName (functionIndex)
cpp	string functionName (int functionIndex)
m	-(NSString*) functionName : (int) functionIndex
pas	string functionName (functionIndex : integer): string
vb	function functionName (ByVal functionIndex As Integer) As String
cs	string functionName (int functionIndex)
java	String functionName (int functionIndex)
py	functionName (functionIndex)
php	function functionName (\$functionIndex)
ts	async functionName (functionIndex : number): Promise<string>
es	async functionName (functionIndex)

Parameters :

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

Returns :

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

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

module→functionType()**YModule**

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

js	function functionType (functionIndex)
cpp	string functionType (int functionIndex)
pas	string functionType (functionIndex : integer): string
vb	function functionType (ByVal functionIndex As Integer) As String
cs	string functionType (int functionIndex)
java	String functionType (int functionIndex)
py	functionType (functionIndex)
php	function functionType (\$ functionIndex)
ts	async functionType (functionIndex : number): Promise<string>
es	async functionType (functionIndex)

Parameters :

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

Returns :

a string corresponding to the type of the function

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

module→functionValue()**YModule**

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

js	function functionValue (functionIndex)
cpp	string functionValue (int functionIndex)
m	-(NSString*) functionValue : (int) functionIndex
pas	string functionValue (functionIndex : integer): string
vb	function functionValue (ByVal functionIndex As Integer) As String
cs	string functionValue (int functionIndex)
java	String functionValue (int functionIndex)
py	functionValue (functionIndex)
php	function functionValue (\$functionIndex)
ts	async functionValue (functionIndex : number): Promise<string>
es	async functionValue (functionIndex)

Parameters :

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

Returns :

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

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

module→**get_allSettings()****YModule****module**→**allSettings()**

Returns all the settings and uploaded files of the module.

js	function get_allSettings ()
cpp	string get_allSettings ()
m	-(NSMutableData*) allSettings
pas	TByteArray get_allSettings (): TByteArray
vb	function get_allSettings () As Byte
cs	byte[] get_allSettings ()
java	byte[] get_allSettings ()
uwp	async Task<byte[]> get_allSettings ()
py	get_allSettings ()
php	function get_allSettings ()
ts	async get_allSettings (): Promise<Uint8Array>
es	async get_allSettings ()
dnp	byte[] get_allSettings ()
cp	string get_allSettings ()
cmd	YModule target get_allSettings

Useful to backup all the logical names, calibrations parameters, and uploaded files of a device.

Returns :

a binary buffer with all the settings.

On failure, throws an exception or returns an binary object of size 0.

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

Returns the state of the localization beacon.

js	function get_beacon ()
cpp	Y_BEACON_enum get_beacon ()
m	-(Y_BEACON_enum) beacon
pas	Integer get_beacon (): Integer
vb	function get_beacon () As Integer
cs	int get_beacon ()
java	int get_beacon ()
uwp	async Task<int> get_beacon ()
py	get_beacon ()
php	function get_beacon ()
ts	async get_beacon (): Promise<YModule_Beacon>
es	async get_beacon ()
dnp	int get_beacon ()
cp	int get_beacon ()
cmd	YModule target get_beacon

Returns :

either YModule.BEACON_OFF or YModule.BEACON_ON, according to the state of the localization beacon

On failure, throws an exception or returns YModule.BEACON_INVALID.

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

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

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

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

Returns :

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

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

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

js	function get_errorType ()
cpp	YRETCODE get_errorType ()
m	-(YRETCODE) errorType
pas	YRETCODE get_errorType (): YRETCODE
vb	function get_errorType () As YRETCODE
cs	YRETCODE get_errorType ()
java	int get_errorType ()
py	get_errorType ()
php	function get_errorType ()
ts	get_errorType (): number
es	get_errorType ()

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

Returns :

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

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

Returns the version of the firmware embedded in the module.

js	function get_firmwareRelease ()
cpp	string get_firmwareRelease ()
m	-(NSString*) firmwareRelease
pas	string get_firmwareRelease (): string
vb	function get_firmwareRelease () As String
cs	string get_firmwareRelease ()
java	String get_firmwareRelease ()
uwp	async Task<string> get_firmwareRelease ()
py	get_firmwareRelease ()
php	function get_firmwareRelease ()
ts	async get_firmwareRelease (): Promise<string>
es	async get_firmwareRelease ()
dnp	string get_firmwareRelease ()
cp	string get_firmwareRelease ()
cmd	YModule target get_firmwareRelease

Returns :

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

On failure, throws an exception or returns YModule.FIRMWARERELEASE_INVALID.

module→get_functionIds()**YModule****module→functionIds()**

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

js	function get_functionIds (funType)
cpp	vector<string> get_functionIds (string funType)
m	-(NSMutableArray*) functionIds : (NSString*) funType
pas	TStringArray get_functionIds (funType : string): TStringArray
vb	function get_functionIds (ByVal funType As String) As List
cs	List<string> get_functionIds (string funType)
java	ArrayList<String> get_functionIds (String funType)
uwp	async Task<List<string>> get_functionIds (string funType)
py	get_functionIds (funType)
php	function get_functionIds (\$ funType)
ts	async get_functionIds (funType : string): Promise<string[]
es	async get_functionIds (funType)
dnp	string[] get_functionIds (string funType)
cp	vector<string> get_functionIds (string funType)
cmd	YModule target get_functionIds funType

Parameters :

funType The type of function (Relay, LightSensor, Voltage,...)

Returns :

an array of strings.

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

Returns the unique hardware identifier of the module.

js	function get_hardwareId ()
cpp	string get_hardwareId ()
m	-(NSString*) hardwareId
vb	function get_hardwareId () As String
cs	string get_hardwareId ()
java	String get_hardwareId ()
py	get_hardwareId ()
php	function get_hardwareId ()
ts	async get_hardwareId (): Promise<string>
es	async get_hardwareId ()
dnp	string get_hardwareId ()
cp	string get_hardwareId ()
pas	string get_hardwareId (): string
uwp	async Task<string> get_hardwareId ()
cmd	YModule target get_hardwareId

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

Returns :

a string that uniquely identifies the module

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

Returns the icon of the module.

js	function get_icon2d ()
cpp	string get_icon2d ()
m	-(NSMutableData*) icon2d
pas	TByteArray get_icon2d (): TByteArray
vb	function get_icon2d () As Byte
cs	byte[] get_icon2d ()
java	byte[] get_icon2d ()
uwp	async Task<byte[]> get_icon2d ()
py	get_icon2d ()
php	function get_icon2d ()
ts	async get_icon2d (): Promise<Uint8Array>
es	async get_icon2d ()
dnp	byte[] get_icon2d ()
cp	string get_icon2d ()
cmd	YModule target get_icon2d

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

Returns :

a binary buffer with module icon, in png format. On failure, throws an exception or returns YAPI_INVALID_STRING.

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

Returns a string with last logs of the module.

js	function get_lastLogs ()
cpp	string get_lastLogs ()
m	-(NSString*) lastLogs
pas	string get_lastLogs (): string
vb	function get_lastLogs () As String
cs	string get_lastLogs ()
java	String get_lastLogs ()
uwp	async Task<string> get_lastLogs ()
py	get_lastLogs ()
php	function get_lastLogs ()
ts	async get_lastLogs (): Promise<string>
es	async get_lastLogs ()
dnp	string get_lastLogs ()
cp	string get_lastLogs ()
cmd	YModule target get_lastLogs

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

Returns :

a string with last logs of the module. On failure, throws an exception or returns YAPI_INVALID_STRING.

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

Returns the logical name of the module.

js	function get_logicalName ()
cpp	string get_logicalName ()
m	-(NSString*) logicalName
pas	string get_logicalName (): string
vb	function get_logicalName () As String
cs	string get_logicalName ()
java	String get_logicalName ()
uwp	async Task<string> get_logicalName ()
py	get_logicalName ()
php	function get_logicalName ()
ts	async get_logicalName (): Promise<string>
es	async get_logicalName ()
dnp	string get_logicalName ()
cp	string get_logicalName ()
cmd	YModule target get_logicalName

Returns :

a string corresponding to the logical name of the module

On failure, throws an exception or returns YModule.LOGICALNAME_INVALID.

module→**get_luminosity()****YModule****module**→**luminosity()**

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

js	function get_luminosity ()
cpp	int get_luminosity ()
m	-(int) luminosity
pas	LongInt get_luminosity (): LongInt
vb	function get_luminosity () As Integer
cs	int get_luminosity ()
java	int get_luminosity ()
uwp	async Task<int> get_luminosity ()
py	get_luminosity ()
php	function get_luminosity ()
ts	async get_luminosity (): Promise<number>
es	async get_luminosity ()
dnp	int get_luminosity ()
cp	int get_luminosity ()
cmd	YModule target get_luminosity

Returns :

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

On failure, throws an exception or returns YModule.LUMINOSITY_INVALID.

module→get_parentHub()**YModule****module→parentHub()**

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

js	function get_parentHub ()
cpp	string get_parentHub ()
m	-(NSString*) parentHub
pas	string get_parentHub (): string
vb	function get_parentHub () As String
cs	string get_parentHub ()
java	String get_parentHub ()
uwp	async Task<string> get_parentHub ()
py	get_parentHub ()
php	function get_parentHub ()
ts	async get_parentHub (): Promise<string>
es	async get_parentHub ()
dnp	string get_parentHub ()
cp	string get_parentHub ()
cmd	YModule target get_parentHub

If the module is connected by USB, or if the module is the root YoctoHub, an empty string is returned.

Returns :

a string with the serial number of the YoctoHub or an empty string

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

Returns the current state of persistent module settings.

js	function get_persistentSettings ()
cpp	Y_PERSISTENTSETTINGS_enum get_persistentSettings ()
m	-(Y_PERSISTENTSETTINGS_enum) persistentSettings
pas	Integer get_persistentSettings (): Integer
vb	function get_persistentSettings () As Integer
cs	int get_persistentSettings ()
java	int get_persistentSettings ()
uwp	async Task<int> get_persistentSettings ()
py	get_persistentSettings ()
php	function get_persistentSettings ()
ts	async get_persistentSettings (): Promise<YModule_PersistentSettings>
es	async get_persistentSettings ()
dnp	int get_persistentSettings ()
cp	int get_persistentSettings ()
cmd	YModule target get_persistentSettings

Returns :

a value among YModule.PERSISTENTSETTINGS_LOADED, YModule.PERSISTENTSETTINGS_SAVED and YModule.PERSISTENTSETTINGS_MODIFIED corresponding to the current state of persistent module settings

On failure, throws an exception or returns YModule.PERSISTENTSETTINGS_INVALID.

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

Returns the USB device identifier of the module.

js	function get_productId ()
cpp	int get_productId ()
m	-(int) productId
pas	LongInt get_productId (): LongInt
vb	function get_productId () As Integer
cs	int get_productId ()
java	int get_productId ()
uwp	async Task<int> get_productId ()
py	get_productId ()
php	function get_productId ()
ts	async get_productId (): Promise<number>
es	async get_productId ()
dnp	int get_productId ()
cp	int get_productId ()
cmd	YModule target get_productId

Returns :

an integer corresponding to the USB device identifier of the module

On failure, throws an exception or returns YModule.PRODUCTID_INVALID.

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

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

js	function get_productName ()
cpp	string get_productName ()
m	-(NSString*) productName
pas	string get_productName (): string
vb	function get_productName () As String
cs	string get_productName ()
java	String get_productName ()
uwp	async Task<string> get_productName ()
py	get_productName ()
php	function get_productName ()
ts	async get_productName (): Promise<string>
es	async get_productName ()
dnp	string get_productName ()
cp	string get_productName ()
cmd	YModule target get_productName

Returns :

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

On failure, throws an exception or returns YModule.PRODUCTNAME_INVALID.

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

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

js	function get_productRelease ()
cpp	int get_productRelease ()
m	-(int) productRelease
pas	LongInt get_productRelease (): LongInt
vb	function get_productRelease () As Integer
cs	int get_productRelease ()
java	int get_productRelease ()
uwp	async Task<int> get_productRelease ()
py	get_productRelease ()
php	function get_productRelease ()
ts	async get_productRelease (): Promise<number>
es	async get_productRelease ()
dnp	int get_productRelease ()
cp	int get_productRelease ()
cmd	YModule target get_productRelease

The original hardware release returns value 1, revision B returns value 2, etc.

Returns :

an integer corresponding to the release number of the module hardware, preprogrammed at the factory

On failure, throws an exception or returns YModule.PRODUCTRELEASE_INVALID.

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

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

js	function get_rebootCountdown ()
cpp	int get_rebootCountdown ()
m	-(int) rebootCountdown
pas	LongInt get_rebootCountdown (): LongInt
vb	function get_rebootCountdown () As Integer
cs	int get_rebootCountdown ()
java	int get_rebootCountdown ()
uwp	async Task<int> get_rebootCountdown ()
py	get_rebootCountdown ()
php	function get_rebootCountdown ()
ts	async get_rebootCountdown (): Promise<number>
es	async get_rebootCountdown ()
dnp	int get_rebootCountdown ()
cp	int get_rebootCountdown ()
cmd	YModule target get_rebootCountdown

Returns :

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

On failure, throws an exception or returns YModule.REBOOTCOUNTDOWN_INVALID.

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

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

js	function get_serialNumber ()
cpp	string get_serialNumber ()
m	-(NSString*) serialNumber
pas	string get_serialNumber (): string
vb	function get_serialNumber () As String
cs	string get_serialNumber ()
java	String get_serialNumber ()
uwp	async Task<string> get_serialNumber ()
py	get_serialNumber ()
php	function get_serialNumber ()
ts	async get_serialNumber (): Promise<string>
es	async get_serialNumber ()
dnp	string get_serialNumber ()
cp	string get_serialNumber ()
cmd	YModule target get_serialNumber

Returns :

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

On failure, throws an exception or returns YModule.SERIALNUMBER_INVALID.

module→**get_subDevices()****YModule****module**→**subDevices()**

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

js	function get_subDevices ()
cpp	vector<string> get_subDevices ()
m	-(NSMutableArray*) subDevices
pas	TStringArray get_subDevices (): TStringArray
vb	function get_subDevices () As List
cs	List<string> get_subDevices ()
java	ArrayList<String> get_subDevices ()
uwp	async Task<List<string>> get_subDevices ()
py	get_subDevices ()
php	function get_subDevices ()
ts	async get_subDevices (): Promise<string[]
es	async get_subDevices ()
dnp	string[] get_subDevices ()
cp	vector<string> get_subDevices ()
cmd	YModule target get_subDevices

This method only makes sense when called for a YoctoHub/VirtualHub. Otherwise, an empty array will be returned.

Returns :

an array of strings containing the sub modules.

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

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

js	function get_upTime ()
cpp	s64 get_upTime ()
m	-(s64) upTime
pas	int64 get_upTime (): int64
vb	function get_upTime () As Long
cs	long get_upTime ()
java	long get_upTime ()
uwp	async Task<long> get_upTime ()
py	get_upTime ()
php	function get_upTime ()
ts	async get_upTime (): Promise<number>
es	async get_upTime ()
dnp	long get_upTime ()
cp	s64 get_upTime ()
cmd	YModule target get_upTime

Returns :

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

On failure, throws an exception or returns YModule.UPTIME_INVALID.

module→**get_url()****YModule****module**→**url()**

Returns the URL used to access the module.

js	function get_url ()
cpp	string get_url ()
m	-(NSString*) url
pas	string get_url (): string
vb	function get_url () As String
cs	string get_url ()
java	String get_url ()
uwp	async Task<string> get_url ()
py	get_url ()
php	function get_url ()
ts	async get_url (): Promise<string>
es	async get_url ()
dnp	string get_url ()
cp	string get_url ()
cmd	YModule target get_url

If the module is connected by USB, the string 'usb' is returned.

Returns :

a string with the URL of the module.

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

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

js	function get_usbCurrent ()
cpp	int get_usbCurrent ()
m	-(int) usbCurrent
pas	LongInt get_usbCurrent (): LongInt
vb	function get_usbCurrent () As Integer
cs	int get_usbCurrent ()
java	int get_usbCurrent ()
uwp	async Task<int> get_usbCurrent ()
py	get_usbCurrent ()
php	function get_usbCurrent ()
ts	async get_usbCurrent (): Promise<number>
es	async get_usbCurrent ()
dnp	int get_usbCurrent ()
cp	int get_usbCurrent ()
cmd	YModule target get_usbCurrent

Returns :

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

On failure, throws an exception or returns YModule.USBCURRENT_INVALID.

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

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

js	function get_userData ()
cpp	void * get_userData ()
m	-(id) userData
pas	Tobject get_userData (): Tobject
vb	function get_userData () As Object
cs	object get_userData ()
java	Object get_userData ()
py	get_userData ()
php	function get_userData ()
ts	async get_userData (): Promise<object null>
es	async get_userData ()

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

Returns :

the object stored previously by the caller.

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

Returns the value previously stored in this attribute.

js	function get_userVar ()
cpp	int get_userVar ()
m	-(int) userVar
pas	LongInt get_userVar (): LongInt
vb	function get_userVar () As Integer
cs	int get_userVar ()
java	int get_userVar ()
uwp	async Task<int> get_userVar ()
py	get_userVar ()
php	function get_userVar ()
ts	async get_userVar (): Promise<number>
es	async get_userVar ()
dnp	int get_userVar ()
cp	int get_userVar ()
cmd	YModule target get_userVar

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

Returns :

an integer corresponding to the value previously stored in this attribute

On failure, throws an exception or returns YModule.USERVAR_INVALID.

module→hasFunction()**YModule**

Tests if the device includes a specific function.

js	function hasFunction (funcId)
cpp	bool hasFunction (string funcId)
m	-(bool) hasFunction : (NSString*) funcId
pas	boolean hasFunction (funcId : string): boolean
vb	function hasFunction (ByVal funcId As String) As Boolean
cs	bool hasFunction (string funcId)
java	boolean hasFunction (String funcId)
uwp	async Task<bool> hasFunction (string funcId)
py	hasFunction (funcId)
php	function hasFunction (\$funcId)
ts	async hasFunction (funcId : string): Promise<boolean>
es	async hasFunction (funcId)
dnp	bool hasFunction (string funcId)
cp	bool hasFunction (string funcId)
cmd	YModule target hasFunction funcId

This method takes a function identifier and returns a boolean.

Parameters :

funcId the requested function identifier

Returns :

true if the device has the function identifier

module→isOnline()**YModule**

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

js	function isOnline ()
cpp	bool isOnline ()
m	-(BOOL) isOnline
pas	boolean isOnline (): boolean
vb	function isOnline () As Boolean
cs	bool isOnline ()
java	boolean isOnline ()
py	isOnline ()
php	function isOnline ()
ts	async isOnline (): Promise<boolean>
es	async isOnline ()
dnp	bool isOnline ()
cp	bool isOnline ()

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

Returns :

`true` if the module can be reached, and `false` otherwise

module→isOnline_async()**YModule**

Checks if the module is currently reachable, without raising any error.

```
js function isOnline_async( callback, context)
```

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking Firefox JavaScript VM that does not implement context switching during blocking I/O calls.

Parameters :

- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving module object and the boolean result
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

module→load()**YModule**

Preloads the module cache with a specified validity duration.

js	function load (msValidity)
cpp	YRETCODE load (int msValidity)
m	-(YRETCODE) load : (u64) msValidity
pas	YRETCODE load (msValidity : u64): YRETCODE
vb	function load (ByVal msValidity As Long) As YRETCODE
cs	YRETCODE load (ulong msValidity)
java	int load (long msValidity)
py	load (msValidity)
php	function load (\$msValidity)
ts	async load (msValidity : number): Promise<number>
es	async load (msValidity)

By default, whenever accessing a device, all module attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded module parameters, in milliseconds

Returns :

YAPI . SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→load_async()**YModule**

Preloads the module cache with a specified validity duration (asynchronous version).

```
js function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all module attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking Firefox JavaScript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous JavaScript calls for more details.

Parameters :

- msValidity** an integer corresponding to the validity of the loaded module parameters, in milliseconds
- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving module object and the error code (or `YAPI.SUCCESS`)
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

module→log()**YModule**

Adds a text message to the device logs.

js	function log (text)
cpp	int log (string text)
m	-(int) log : (NSString*) text
pas	LongInt log (text : string): LongInt
vb	function log (ByVal text As String) As Integer
cs	int log (string text)
java	int log (String text)
uwp	async Task<int> log (string text)
py	log (text)
php	function log (\$ text)
ts	async log (text : string): Promise<number>
es	async log (text)
dnp	int log (string text)
cp	int log (string text)
cmd	YModule target log text

This function is useful in particular to trace the execution of HTTP callbacks. If a newline is desired after the message, it must be included in the string.

Parameters :

text the string to append to the logs.

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→nextModule()**YModule**

Continues the module enumeration started using `yFirstModule()`.

js	function nextModule ()
cpp	YModule * nextModule ()
m	-(nullable YModule*) nextModule
pas	TYModule nextModule (): TYModule
vb	function nextModule () As YModule
cs	YModule nextModule ()
java	YModule nextModule ()
uwp	YModule nextModule ()
py	nextModule ()
php	function nextModule ()
ts	nextModule (): YModule null
es	nextModule ()

Caution: You can't make any assumption about the returned modules order. If you want to find a specific module, use `Module.findModule()` and a hardwareID or a logical name.

Returns :

a pointer to a `YModule` object, corresponding to the next module found, or a `null` pointer if there are no more modules to enumerate.

module→reboot()**YModule**

Schedules a simple module reboot after the given number of seconds.

js	function reboot (secBeforeReboot)
cpp	int reboot (int secBeforeReboot)
m	-(int) reboot : (int) secBeforeReboot
pas	LongInt reboot (secBeforeReboot : LongInt): LongInt
vb	function reboot (ByVal secBeforeReboot As Integer) As Integer
cs	int reboot (int secBeforeReboot)
java	int reboot (int secBeforeReboot)
uwp	async Task<int> reboot (int secBeforeReboot)
py	reboot (secBeforeReboot)
php	function reboot (\$secBeforeReboot)
ts	async reboot (secBeforeReboot : number): Promise<number>
es	async reboot (secBeforeReboot)
dnp	int reboot (int secBeforeReboot)
cp	int reboot (int secBeforeReboot)
cmd	YModule target reboot secBeforeReboot

Parameters :

secBeforeReboot number of seconds before rebooting

Returns :

YAPI . SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→registerBeaconCallback()**YModule**

Register a callback function, to be called when the localization beacon of the module has been changed.

js	function registerBeaconCallback (callback)
cpp	int registerBeaconCallback (YModuleBeaconCallback callback)
m	-(int) registerBeaconCallback : (YModuleBeaconCallback _Nullable) callback
pas	LongInt registerBeaconCallback (callback : TYModuleBeaconCallback): LongInt
vb	function registerBeaconCallback (ByVal callback As YModuleBeaconCallback) As Integer
cs	int registerBeaconCallback (BeaconCallback callback)
java	int registerBeaconCallback (BeaconCallback callback)
uwp	async Task<int> registerBeaconCallback (BeaconCallback callback)
py	registerBeaconCallback (callback)
php	function registerBeaconCallback (\$callback)
ts	async registerBeaconCallback (callback : YModuleBeaconCallback null): Promise<number>
es	async registerBeaconCallback (callback)

The callback function should take two arguments: the YModule object of which the beacon has changed, and an integer describing the new beacon state.

Parameters :

callback The callback function to call, or `null` to unregister a

module→registerConfigChangeCallback()**YModule**

Register a callback function, to be called when a persistent settings in a device configuration has been changed (e.g.

js	function registerConfigChangeCallback (callback)
cpp	int registerConfigChangeCallback (YModuleConfigChangeCallback callback)
m	-(int) registerConfigChangeCallback : (YModuleConfigChangeCallback _Nullable) callback
pas	LongInt registerConfigChangeCallback (callback : TYModuleConfigChangeCallback): LongInt
vb	function registerConfigChangeCallback (ByVal callback As YModuleConfigChangeCallback) As Integer
cs	int registerConfigChangeCallback (ConfigChangeCallback callback)
java	int registerConfigChangeCallback (ConfigChangeCallback callback)
uwp	async Task<int> registerConfigChangeCallback (ConfigChangeCallback callback)
py	registerConfigChangeCallback (callback)
php	function registerConfigChangeCallback (\$callback)
ts	async registerConfigChangeCallback (callback : YModuleConfigChangeCallback null): Promise<number>
es	async registerConfigChangeCallback (callback)

change of unit, etc).

Parameters :

callback a procedure taking a YModule parameter, or null

module→registerLogCallback()**YModule**

Registers a device log callback function.

js	function registerLogCallback (callback)
cpp	int registerLogCallback (YModuleLogCallback callback)
m	-(int) registerLogCallback : (YModuleLogCallback _Nullable) callback
pas	LongInt registerLogCallback (callback : TYModuleLogCallback): LongInt
vb	function registerLogCallback (ByVal callback As YModuleLogCallback) As Integer
cs	int registerLogCallback (LogCallback callback)
java	int registerLogCallback (LogCallback callback)
uwp	async Task<int> registerLogCallback (LogCallback callback)
py	registerLogCallback (callback)
php	function registerLogCallback (\$callback)
ts	async registerLogCallback (callback : YModuleLogCallback null): Promise<number>
es	async registerLogCallback (callback)

This callback will be called each time that a module sends a new log message. Mostly useful to debug a Yoctopuce module.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the module object that emitted the log message, and the character string containing the log.

module→revertFromFlash()**YModule**

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

js	function revertFromFlash ()
cpp	int revertFromFlash ()
m	-(int) revertFromFlash
pas	LongInt revertFromFlash (): LongInt
vb	function revertFromFlash () As Integer
cs	int revertFromFlash ()
java	int revertFromFlash ()
uwp	async Task<int> revertFromFlash ()
py	revertFromFlash ()
php	function revertFromFlash ()
ts	async revertFromFlash (): Promise<number>
es	async revertFromFlash ()
dnp	int revertFromFlash ()
cp	int revertFromFlash ()
cmd	YModule target revertFromFlash

Returns :

YAPI . SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→saveToFlash()**YModule**

Saves current settings in the nonvolatile memory of the module.

js	function saveToFlash ()
cpp	int saveToFlash ()
m	-(int) saveToFlash
pas	LongInt saveToFlash (): LongInt
vb	function saveToFlash () As Integer
cs	int saveToFlash ()
java	int saveToFlash ()
uwp	async Task<int> saveToFlash ()
py	saveToFlash ()
php	function saveToFlash ()
ts	async saveToFlash (): Promise<number>
es	async saveToFlash ()
dnp	int saveToFlash ()
cp	int saveToFlash ()
cmd	YModule target saveToFlash

Warning: the number of allowed save operations during a module life is limited (about 100000 cycles). Do not call this function within a loop.

Returns :

YAPI . SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_allSettings()**YModule****module→setAllSettings()**

Restores all the settings of the device.

js	function set_allSettings (settings)
cpp	int set_allSettings (string settings)
m	-(int) setAllSettings : (NSData*) settings
pas	LongInt set_allSettings (settings : TByteArray): LongInt
vb	procedure set_allSettings (ByVal settings As Byte())
cs	int set_allSettings (byte[] settings)
java	int set_allSettings (byte[] settings)
uwp	async Task<int> set_allSettings (byte[] settings)
py	set_allSettings (settings)
php	function set_allSettings (\$ settings)
ts	async set_allSettings (settings : Uint8Array): Promise<number>
es	async set_allSettings (settings)
dnp	int set_allSettings (byte[] settings)
cp	int set_allSettings (string settings)
cmd	YModule target set_allSettings settings

Useful to restore all the logical names and calibrations parameters of a module from a backup. Remember to call the `saveToFlash()` method of the module if the modifications must be kept.

Parameters :

settings a binary buffer with all the settings.

Returns :

YAPI . SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→**set_allSettingsAndFiles()****YModule****module**→**setAllSettingsAndFiles()**

Restores all the settings and uploaded files to the module.

js	function set_allSettingsAndFiles (settings)
cpp	int set_allSettingsAndFiles (string settings)
m	-(int) setAllSettingsAndFiles : (NSData*) settings
pas	LongInt set_allSettingsAndFiles (settings : TByteArray): LongInt
vb	procedure set_allSettingsAndFiles (ByVal settings As Byte())
cs	int set_allSettingsAndFiles (byte[] settings)
java	int set_allSettingsAndFiles (byte[] settings)
uwp	async Task<int> set_allSettingsAndFiles (byte[] settings)
py	set_allSettingsAndFiles (settings)
php	function set_allSettingsAndFiles (\$ settings)
ts	async set_allSettingsAndFiles (settings : Uint8Array): Promise<number>
es	async set_allSettingsAndFiles (settings)
dnp	int set_allSettingsAndFiles (byte[] settings)
cp	int set_allSettingsAndFiles (string settings)
cmd	YModule target set_allSettingsAndFiles settings

This method is useful to restore all the logical names and calibrations parameters, uploaded files etc. of a device from a backup. Remember to call the `saveToFlash()` method of the module if the modifications must be kept.

Parameters :

settings a binary buffer with all the settings.

Returns :

YAPI . SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→**set_beacon()****YModule****module**→**setBeacon()**

Turns on or off the module localization beacon.

js	function set_beacon (newval)
cpp	int set_beacon (Y_BEACON_enum newval)
m	-(int) setBeacon : (Y_BEACON_enum) newval
pas	integer set_beacon (newval : Integer): integer
vb	function set_beacon (ByVal newval As Integer) As Integer
cs	int set_beacon (int newval)
java	int set_beacon (int newval)
uwp	async Task<int> set_beacon (int newval)
py	set_beacon (newval)
php	function set_beacon (\$newval)
ts	async set_beacon (newval : YModule_Beacon): Promise<number>
es	async set_beacon (newval)
dnp	int set_beacon (int newval)
cp	int set_beacon (int newval)
cmd	YModule target set_beacon newval

Parameters :

newval either YModule.BEACON_OFF or YModule.BEACON_ON

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→**set_logicalName()****YModule****module**→**setLogicalName()**

Changes the logical name of the module.

js	function set_logicalName (newval)
cpp	int set_logicalName (string newval)
m	-(int) setLogicalName : (NSString*) newval
pas	integer set_logicalName (newval : string): integer
vb	function set_logicalName (ByVal newval As String) As Integer
cs	int set_logicalName (string newval)
java	int set_logicalName (String newval)
uwp	async Task<int> set_logicalName (string newval)
py	set_logicalName (newval)
php	function set_logicalName (\$ newval)
ts	async set_logicalName (newval : string): Promise<number>
es	async set_logicalName (newval)
dnp	int set_logicalName (string newval)
cp	int set_logicalName (string newval)
cmd	YModule target set_logicalName newval

You can use `yCheckLogicalName( )` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the module

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→**set_luminosity()****YModule****module**→**setLuminosity()**

Changes the luminosity of the module informative leds.

js	function set_luminosity (newval)
cpp	int set_luminosity (int newval)
m	-(int) setLuminosity : (int) newval
pas	integer set_luminosity (newval : LongInt): integer
vb	function set_luminosity (ByVal newval As Integer) As Integer
cs	int set_luminosity (int newval)
java	int set_luminosity (int newval)
uwp	async Task<int> set_luminosity (int newval)
py	set_luminosity (newval)
php	function set_luminosity (\$newval)
ts	async set_luminosity (newval : number): Promise<number>
es	async set_luminosity (newval)
dnp	int set_luminosity (int newval)
cp	int set_luminosity (int newval)
cmd	YModule target set_luminosity newval

The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer corresponding to the luminosity of the module informative leds

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→**set_userData()****YModule****module**→**setUserData()**

Stores a user context provided as argument in the userData attribute of the function.

js	function set_userData (data)
cpp	void set_userData (void * data)
m	-(void) setUserData : (id) data
pas	set_userData (data : Tobject)
vb	procedure set_userData (ByVal data As Object)
cs	void set_userData (object data)
java	void set_userData (Object data)
py	set_userData (data)
php	function set_userData (\$ data)
ts	async set_userData (data : object null): Promise<void>
es	async set_userData (data)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

module→set_userVar()**YModule****module→setUserVar()**

Stores a 32 bit value in the device RAM.

js	function set_userVar (newval)
cpp	int set_userVar (int newval)
m	-(int) setUserVar : (int) newval
pas	integer set_userVar (newval : LongInt): integer
vb	function set_userVar (ByVal newval As Integer) As Integer
cs	int set_userVar (int newval)
java	int set_userVar (int newval)
uwp	async Task<int> set_userVar (int newval)
py	set_userVar (newval)
php	function set_userVar (\$newval)
ts	async set_userVar (newval : number): Promise<number>
es	async set_userVar (newval)
dnp	int set_userVar (int newval)
cp	int set_userVar (int newval)
cmd	YModule target set_userVar newval

This attribute is at programmer disposal, should he need to store a state variable. On startup and after a device reboot, the value is always reset to zero.

Parameters :

newval an integer

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→triggerConfigChangeCallback()**YModule**

Triggers a configuration change callback, to check if they are supported or not.

js	function triggerConfigChangeCallback ()
cpp	int triggerConfigChangeCallback ()
m	-(int) triggerConfigChangeCallback
pas	LongInt triggerConfigChangeCallback (): LongInt
vb	function triggerConfigChangeCallback () As Integer
cs	int triggerConfigChangeCallback ()
java	int triggerConfigChangeCallback ()
uwp	async Task<int> triggerConfigChangeCallback ()
py	triggerConfigChangeCallback ()
php	function triggerConfigChangeCallback ()
ts	async triggerConfigChangeCallback (): Promise<number>
es	async triggerConfigChangeCallback ()
dnp	int triggerConfigChangeCallback ()
cp	int triggerConfigChangeCallback ()
cmd	YModule target triggerConfigChangeCallback

module→triggerFirmwareUpdate()

YModule

Schedules a module reboot into special firmware update mode.

js	function triggerFirmwareUpdate (secBeforeReboot)
cpp	int triggerFirmwareUpdate (int secBeforeReboot)
m	-(int) triggerFirmwareUpdate : (int) secBeforeReboot
pas	LongInt triggerFirmwareUpdate (secBeforeReboot : LongInt): LongInt
vb	function triggerFirmwareUpdate (ByVal secBeforeReboot As Integer) As Integer
cs	int triggerFirmwareUpdate (int secBeforeReboot)
java	int triggerFirmwareUpdate (int secBeforeReboot)
uwp	async Task<int> triggerFirmwareUpdate (int secBeforeReboot)
py	triggerFirmwareUpdate (secBeforeReboot)
php	function triggerFirmwareUpdate (\$secBeforeReboot)
ts	async triggerFirmwareUpdate (secBeforeReboot : number): Promise<number>
es	async triggerFirmwareUpdate (secBeforeReboot)
dnp	int triggerFirmwareUpdate (int secBeforeReboot)
cp	int triggerFirmwareUpdate (int secBeforeReboot)
cmd	YModule target triggerFirmwareUpdate secBeforeReboot

Parameters :

secBeforeReboot number of seconds before rebooting

Returns :

YAPI . SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→updateFirmware()**YModule**

Prepares a firmware update of the module.

js	function updateFirmware (path)
cpp	YFirmwareUpdate updateFirmware (string path)
m	-(YFirmwareUpdate*) updateFirmware : (NSString*) path
pas	TYFirmwareUpdate updateFirmware (path : string): TYFirmwareUpdate
vb	function updateFirmware (ByVal path As String) As YFirmwareUpdate
cs	YFirmwareUpdate updateFirmware (string path)
java	YFirmwareUpdate updateFirmware (String path)
uwp	async Task<YFirmwareUpdate> updateFirmware (string path)
py	updateFirmware (path)
php	function updateFirmware (\$path)
ts	async updateFirmware (path : string): Promise<YFirmwareUpdate>
es	async updateFirmware (path)
dnp	YFirmwareUpdateProxy updateFirmware (string path)
cp	YFirmwareUpdateProxy* updateFirmware (string path)
cmd	YModule target updateFirmware path

This method returns a YFirmwareUpdate object which handles the firmware update process.

Parameters :

path the path of the .byn file to use.

Returns :

a YFirmwareUpdate object or NULL on error.

module→updateFirmwareEx()**YModule**

Prepares a firmware update of the module.

js	function updateFirmwareEx (path , force)
cpp	YFirmwareUpdate updateFirmwareEx (string path , bool force)
m	-(YFirmwareUpdate*) updateFirmwareEx : (NSString*) path : (bool) force
pas	TYFirmwareUpdate updateFirmwareEx (path : string, force : boolean): TYFirmwareUpdate
vb	function updateFirmwareEx (ByVal path As String, ByVal force As Boolean) As YFirmwareUpdate
cs	YFirmwareUpdate updateFirmwareEx (string path , bool force)
java	YFirmwareUpdate updateFirmwareEx (String path , boolean force)
uwp	async Task<YFirmwareUpdate> updateFirmwareEx (string path , bool force)
py	updateFirmwareEx (path , force)
php	function updateFirmwareEx (\$path , \$force)
ts	async updateFirmwareEx (path : string, force : boolean): Promise<YFirmwareUpdate>
es	async updateFirmwareEx (path , force)
dnp	YFirmwareUpdateProxy updateFirmwareEx (string path , bool force)
cp	YFirmwareUpdateProxy* updateFirmwareEx (string path , bool force)
cmd	YModule target updateFirmwareEx path force

This method returns a YFirmwareUpdate object which handles the firmware update process.

Parameters :

path the path of the .byn file to use.

force true to force the firmware update even if some prerequisites appear not to be met

Returns :

a YFirmwareUpdate object or NULL on error.

module→wait_async()**YModule**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

js	<code>function wait_async(callback, context)</code>

ts	<code>wait_async(callback: Function, context: object)</code>

es	<code>wait_async(callback, context)</code>

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

24.3. Class YPower

Electrical power sensor control interface, available for instance in the Yocto-Watt

The YPower class allows you to read and configure Yoctopuce electrical power sensors. It inherits from YSensor class the core functions to read measurements, to register callback functions, and to access the autonomous datalogger. This class adds the ability to access the energy counter and the power factor.

In order to use the functions described here, you should include:

es	in HTML: <code><script src='.././lib/yocto_power.js'></script></code> in node.js: <code>require('yoctolib-es2017/yocto_power.js');</code>
js	<code><script type='text/javascript' src='yocto_power.js'></script></code>
cpp	<code>#include "yocto_power.h"</code>
m	<code>#import "yocto_power.h"</code>
pas	<code>uses yocto_power;</code>
vb	<code>yocto_power.vb</code>
cs	<code>yocto_power.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YPower;</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YPower;</code>
py	<code>from yocto_power import *</code>
php	<code>require_once('yocto_power.php');</code>
ts	in HTML: <code>import { YPower } from '.././dist/esm/yocto_power.js';</code> in Node.js: <code>import { YPower } from 'yoctolib-cjs/yocto_power.js';</code>
dnf	<code>import YoctoProxyAPI.YPowerProxy</code>
cp	<code>#include "yocto_power_proxy.h"</code>
vi	<code>YPower.vi</code>
mL	<code>import YoctoProxyAPI.YPowerProxy</code>

Global functions

YPower.FindPower(func)

Retrieves a electrical power sensor for a given identifier.

YPower.FindPowerInContext(yctx, func)

Retrieves a electrical power sensor for a given identifier in a YAPI context.

YPower.FirstPower()

Starts the enumeration of electrical power sensors currently accessible.

YPower.FirstPowerInContext(yctx)

Starts the enumeration of electrical power sensors currently accessible.

YPower.GetSimilarFunctions()

Enumerates all functions of type Power available on the devices currently reachable by the library, and returns their unique hardware ID.

YPower properties

power→AdvMode [writable]

Measuring mode used for the advertised value pushed to the parent hub.

power→AdvertisedValue [read-only]

Short string representing the current state of the function.

power→FriendlyName [read-only]

Global identifier of the function in the format `MODULE_NAME . FUNCTION_NAME`.

power→FunctionId *[read-only]*

Hardware identifier of the sensor, without reference to the module.

power→HardwareId *[read-only]*

Unique hardware identifier of the function in the form SERIAL . FUNCTIONID.

power→IsOnline *[read-only]*

Checks if the function is currently reachable.

power→LogFrequency *[writable]*

Datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

power→LogicalName *[writable]*

Logical name of the function.

power→ReportFrequency *[writable]*

Timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

power→Resolution *[writable]*

Resolution of the measured values.

power→SerialNumber *[read-only]*

Serial number of the module, as set by the factory.

YPower methods

power→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

power→clearCache()

Invalidates the cache.

power→describe()

Returns a short text that describes unambiguously the instance of the electrical power sensor in the form TYPE (NAME)=SERIAL . FUNCTIONID.

power→get_advMode()

Returns the measuring mode used for the advertised value pushed to the parent hub.

power→get_advertisedValue()

Returns the current value of the electrical power sensor (no more than 6 characters).

power→get_cosPhi()

Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).

power→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in Watt, as a floating point number.

power→get_currentValue()

Returns the current value of the electrical power, in Watt, as a floating point number.

power→get_dataLogger()

Returns the YData logger object of the device hosting the sensor.

power→get_deliveredEnergyMeter()

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time, but only when positive.

power→get_errorMessage()

Returns the error message of the latest error with the electrical power sensor.

power→get_errorType()

Returns the numerical error code of the latest error with the electrical power sensor.

power→get_friendlyName()

Returns a global identifier of the electrical power sensor in the format `MODULE_NAME.FUNCTION_NAME`.

power→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

power→get_functionId()

Returns the hardware identifier of the electrical power sensor, without reference to the module.

power→get_hardwareId()

Returns the unique hardware identifier of the electrical power sensor in the form `SERIAL.FUNCTIONID`.

power→get_highestValue()

Returns the maximal value observed for the electrical power since the device was started.

power→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

power→get_logicalName()

Returns the logical name of the electrical power sensor.

power→get_lowestValue()

Returns the minimal value observed for the electrical power since the device was started.

power→get_meter()

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time, but only when positive.

power→get_meterTimer()

Returns the elapsed time since last energy counter reset, in seconds.

power→get_module()

Gets the `YModule` object for the device on which the function is located.

power→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

power→get_receivedEnergyMeter()

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time, but only when negative.

power→get_recordedData(startTime, endTime)

Retrieves a `YDataSet` object holding historical data for this sensor, for a specified time interval.

power→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

power→get_resolution()

Returns the resolution of the measured values.

power→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

power→get_serialNumber()

Returns the serial number of the module, as set by the factory.

power→get_unit()

Returns the measuring unit for the electrical power.

power→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

power→isOnline()

Checks if the electrical power sensor is currently reachable, without raising any error.

power→isOnline_async(callback, context)

Checks if the electrical power sensor is currently reachable, without raising any error (asynchronous version).

power→isReadOnly()

Test if the function is readOnly.

power→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

power→load(msValidity)

Preloads the electrical power sensor cache with a specified validity duration.

power→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.

power→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

power→load_async(msValidity, callback, context)

Preloads the electrical power sensor cache with a specified validity duration (asynchronous version).

power→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

power→nextPower()

Continues the enumeration of electrical power sensors started using `yFirstPower()`.

power→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

power→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

power→reset()

Resets the energy counters.

power→set_advMode(newval)

Changes the measuring mode used for the advertised value pushed to the parent hub.

power→set_highestValue(newval)

Changes the recorded maximal value observed.

power→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

power→set_logicalName(newval)

Changes the logical name of the electrical power sensor.

power→set_lowestValue(newval)

Changes the recorded minimal value observed.

power→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

power→set_resolution(newval)

Changes the resolution of the measured physical values.

power→set_userData(data)

Stores a user context provided as argument in the `userData` attribute of the function.

power→startDataLogger()

Starts the data logger on the device.

power→stopDataLogger()

Stops the datalogger on the device.

power→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

power→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YPower.FindPower()**YPower****YPower.FindPower()**

Retrieves a electrical power sensor for a given identifier.

js	<code>function yFindPower(func)</code>
cpp	<code>YPower* FindPower(string func)</code>
m	<code>+(YPower*) FindPower : (NSString*) func</code>
pas	<code>TYPower yFindPower(func: string): TYPower</code>
vb	<code>function FindPower(ByVal func As String) As YPower</code>
cs	<code>static YPower FindPower(string func)</code>
java	<code>static YPower FindPower(String func)</code>
uwp	<code>static YPower FindPower(string func)</code>
py	<code>FindPower(func)</code>
php	<code>function FindPower(\$func)</code>
ts	<code>static FindPower(func: string): YPower</code>
es	<code>static FindPower(func)</code>
dnp	<code>static YPowerProxy FindPower(string func)</code>
cp	<code>static YPowerProxy * FindPower(string func)</code>

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the electrical power sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPower.isOnline()` to test if the electrical power sensor is indeed online at a given time. In case of ambiguity when looking for a electrical power sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

If a call to this object's `is_online()` method returns `FALSE` although you are certain that the matching device is plugged, make sure that you did call `registerHub()` at application initialization time.

Parameters :

func a string that uniquely characterizes the electrical power sensor, for instance `YWATTMK1.power`.

Returns :

a `YPower` object allowing you to drive the electrical power sensor.

YPower.FindPowerInContext()

YPower.FindPowerInContext()

YPower

Retrieves a electrical power sensor for a given identifier in a YAPI context.

java	static YPower FindPowerInContext (YAPIContext yctx , String func)
uwp	static YPower FindPowerInContext (YAPIContext yctx , string func)
ts	static FindPowerInContext (yctx : YAPIContext, func : string): YPower
es	static FindPowerInContext (yctx , func)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the electrical power sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YPower.isOnline()` to test if the electrical power sensor is indeed online at a given time. In case of ambiguity when looking for a electrical power sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the electrical power sensor, for instance `YWATTMK1.power`.

Returns :

a YPower object allowing you to drive the electrical power sensor.

YPower.FirstPower()**YPower****YPower.FirstPower()**

Starts the enumeration of electrical power sensors currently accessible.

js	function yFirstPower ()
cpp	YPower * FirstPower ()
m	+(YPower*) FirstPower
pas	TYPower yFirstPower (): TYPower
vb	function FirstPower () As YPower
cs	static YPower FirstPower ()
java	static YPower FirstPower ()
uwp	static YPower FirstPower ()
py	FirstPower ()
php	function FirstPower ()
ts	static FirstPower (): YPower null
es	static FirstPower ()

Use the method `YPower.nextPower()` to iterate on next electrical power sensors.

Returns :

a pointer to a `YPower` object, corresponding to the first electrical power sensor currently online, or a `null` pointer if there are none.

YPower.FirstPowerInContext()**YPower****YPower.FirstPowerInContext()**

Starts the enumeration of electrical power sensors currently accessible.

java	static YPower FirstPowerInContext (YAPIContext yctx)
uwp	static YPower FirstPowerInContext (YAPIContext yctx)
ts	static FirstPowerInContext (yctx : YAPIContext): YPower null
es	static FirstPowerInContext (yctx)

Use the method `YPower . nextPower ( )` to iterate on next electrical power sensors.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YPower` object, corresponding to the first electrical power sensor currently online, or a `null` pointer if there are none.

YPower.GetSimilarFunctions()**YPower****YPower.GetSimilarFunctions()**

Enumerates all functions of type Power available on the devices currently reachable by the library, and returns their unique hardware ID.

dnp	<code>static new string[] GetSimilarFunctions()</code>

cp	<code>static vector<string> GetSimilarFunctions()</code>

Each of these IDs can be provided as argument to the method `YPower.FindPower` to obtain an object that can control the corresponding device.

Returns :

an array of strings, each string containing the unique hardwareId of a device function currently connected.

power→AdvMode**YPower**

Measuring mode used for the advertised value pushed to the parent hub.

dnf

`int AdvMode`

Writable. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

power→**AdvertisedValue****YPower**

Short string representing the current state of the function.

dnp string **AdvertisedValue**

power→FriendlyName**YPower**

Global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

`dnv` `string` **FriendlyName**

The returned string uses the logical names of the module and of the function if they are defined, otherwise the serial number of the module and the hardware identifier of the function (for example: `MyCustomName.relay1`)

power→**FunctionId****YPower**

Hardware identifier of the sensor, without reference to the module.

dnf [string](#) **FunctionId**

For example `re lay1`

power→HardwareId**YPower**

Unique hardware identifier of the function in the form SERIAL . FUNCTIONID.

dnp `string` **HardwareId**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example RELAYL01-123456 . r e l a y 1).

power→IsOnline**YPower**

Checks if the function is currently reachable.

dnf

bool IsOnline

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the function.

power→LogFrequency**YPower**

Datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

dnf

string LogFrequency

Writable. Changes the datalogger recording frequency for this function. The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF". Note that setting the datalogger recording frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

power→LogicalName**YPower**

Logical name of the function.

dnf `string LogicalName`

Writable. You can use `yCheckLogicalName( )` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

power→ReportFrequency**YPower**

Timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

dnf `string ReportFrequency`

Writable. Changes the timed value notification frequency for this function. The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (e.g. "4/h"). To disable timed value notifications for this function, use the value "OFF". Note that setting the timed value notification frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

power→Resolution**YPower**

Resolution of the measured values.

`double Resolution`

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Writable. Changes the resolution of the measured physical values. The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

power→SerialNumber**YPower**

Serial number of the module, as set by the factory.

dnp

 string **SerialNumber**

power→calibrateFromPoints()**YPower**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

js	function calibrateFromPoints (rawValues , refValues)
cpp	int calibrateFromPoints (vector<double> rawValues , vector<double> refValues)
m	-(int) calibrateFromPoints : (NSMutableArray*) rawValues : (NSMutableArray*) refValues
pas	LongInt calibrateFromPoints (rawValues : TDoubleArray, refValues : TDoubleArray): LongInt
vb	procedure calibrateFromPoints (ByVal rawValues As List(Of)
cs	int calibrateFromPoints (List<double> rawValues , List<double> refValues)
java	int calibrateFromPoints (ArrayList<Double> rawValues , ArrayList<Double> refValues)
uwp	async Task<int> calibrateFromPoints (List<double> rawValues , List<double> refValues)
py	calibrateFromPoints (rawValues , refValues)
php	function calibrateFromPoints (\$ rawValues , \$ refValues)
ts	async calibrateFromPoints (rawValues : number[], refValues : number[]): Promise<number>
es	async calibrateFromPoints (rawValues , refValues)
dnp	int calibrateFromPoints (double[] rawValues , double[] refValues)
cp	int calibrateFromPoints (vector<double> rawValues , vector<double> refValues)
cmd	YPower target calibrateFromPoints rawValues refValues

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→clearCache()**YPower**

Invalidates the cache.

js	function clearCache ()
cpp	void clearCache ()
m	-(void) clearCache
pas	clearCache ()
vb	procedure clearCache ()
cs	void clearCache ()
java	void clearCache ()
py	clearCache ()
php	function clearCache ()
ts	async clearCache (): Promise<void>
es	async clearCache ()

Invalidates the cache of the electrical power sensor attributes. Forces the next call to `get_xxx()` or `loadxxx()` to use values that come from the device.

power→describe()**YPower**

Returns a short text that describes unambiguously the instance of the electrical power sensor in the form `TYPE (NAME)=SERIAL . FUNCTIONID`.

js	function describe ()
cpp	string describe ()
m	-(NSString*) describe
pas	string describe (): string
vb	function describe () As String
cs	string describe ()
java	String describe ()
py	describe ()
php	function describe ()
ts	async describe (): Promise<string>
es	async describe ()

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYL01-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the electrical power sensor (ex:
`Relay(MyCustomName.relay1)=RELAYL01-123456.relay1`)

power→get_advMode()**YPower****power→advMode()**

Returns the measuring mode used for the advertised value pushed to the parent hub.

js	function get_advMode ()
cpp	Y_ADVMODE_enum get_advMode ()
m	-(Y_ADVMODE_enum) advMode
pas	Integer get_advMode (): Integer
vb	function get_advMode () As Integer
cs	int get_advMode ()
java	int get_advMode ()
uwp	async Task<int> get_advMode ()
py	get_advMode ()
php	function get_advMode ()
ts	async get_advMode (): Promise<YSensor_AdvMode>
es	async get_advMode ()
dnp	int get_advMode ()
cp	int get_advMode ()
cmd	YPower target get_advMode

Returns :

a value among YPower.ADVMODE_IMMEDIATE, YPower.ADVMODE_PERIOD_AVG, YPower.ADVMODE_PERIOD_MIN and YPower.ADVMODE_PERIOD_MAX corresponding to the measuring mode used for the advertised value pushed to the parent hub

On failure, throws an exception or returns YPower.ADVMODE_INVALID.

power→get_advertisedValue()**YPower****power→advertisedValue()**

Returns the current value of the electrical power sensor (no more than 6 characters).

js	function get_advertisedValue ()
cpp	string get_advertisedValue ()
m	-(NSString*) advertisedValue
pas	string get_advertisedValue (): string
vb	function get_advertisedValue () As String
cs	string get_advertisedValue ()
java	String get_advertisedValue ()
uwp	async Task<string> get_advertisedValue ()
py	get_advertisedValue ()
php	function get_advertisedValue ()
ts	async get_advertisedValue (): Promise<string>
es	async get_advertisedValue ()
dnp	string get_advertisedValue ()
cp	string get_advertisedValue ()
cmd	YPower target get_advertisedValue

Returns :

a string corresponding to the current value of the electrical power sensor (no more than 6 characters).

On failure, throws an exception or returns YPower . ADVERTISEDVALUE_INVALID.

power→get_cosPhi()**YPower****power→cosPhi()**

Returns the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA).

js	function get_cosPhi ()
cpp	double get_cosPhi ()
m	-(double) cosPhi
pas	double get_cosPhi (): double
vb	function get_cosPhi () As Double
cs	double get_cosPhi ()
java	double get_cosPhi ()
uwp	async Task<double> get_cosPhi ()
py	get_cosPhi ()
php	function get_cosPhi ()
ts	async get_cosPhi (): Promise<number>
es	async get_cosPhi ()
dnp	double get_cosPhi ()
cp	double get_cosPhi ()
cmd	YPower target get_cosPhi

Returns :

a floating point number corresponding to the power factor (the ratio between the real power consumed, measured in W, and the apparent power provided, measured in VA)

On failure, throws an exception or returns YPower . COSPHI_INVALID.

power→get_currentRawValue()**YPower****power→currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor, in Watt, as a floating point number.

js	function get_currentRawValue ()
cpp	double get_currentRawValue ()
m	-(double) currentRawValue
pas	double get_currentRawValue (): double
vb	function get_currentRawValue () As Double
cs	double get_currentRawValue ()
java	double get_currentRawValue ()
uwp	async Task<double> get_currentRawValue ()
py	get_currentRawValue ()
php	function get_currentRawValue ()
ts	async get_currentRawValue (): Promise<number>
es	async get_currentRawValue ()
dnp	double get_currentRawValue ()
cp	double get_currentRawValue ()
cmd	YPower target get_currentRawValue

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in Watt, as a floating point number

On failure, throws an exception or returns YPower . CURRENTRAWVALUE_INVALID.

power→**get_currentValue()****YPower****power**→**currentValue()**

Returns the current value of the electrical power, in Watt, as a floating point number.

js	function get_currentValue ()
cpp	double get_currentValue ()
m	-(double) currentValue
pas	double get_currentValue (): double
vb	function get_currentValue () As Double
cs	double get_currentValue ()
java	double get_currentValue ()
uwp	async Task<double> get_currentValue ()
py	get_currentValue ()
php	function get_currentValue ()
ts	async get_currentValue (): Promise<number>
es	async get_currentValue ()
dnp	double get_currentValue ()
cp	double get_currentValue ()
cmd	YPower target get_currentValue

Note that a `get_currentValue()` call will **not** start a measure in the device, it will just return the last measure that occurred in the device. Indeed, internally, each Yoctopuce devices is continuously making measurements at a hardware specific frequency.

If continuously calling `get_currentValue()` leads you to performances issues, then you might consider to switch to callback programming model. Check the "advanced programming" chapter in in your device user manual for more information.

Returns :

a floating point number corresponding to the current value of the electrical power, in Watt, as a floating point number

On failure, throws an exception or returns `YPower.CURRENTVALUE_INVALID`.

power→get_dataLogger()**YPower****power→dataLogger()**

Returns the YDataLogger object of the device hosting the sensor.

js	function get_dataLogger ()
cpp	YDataLogger* get_dataLogger ()
m	-(YDataLogger*) dataLogger
pas	TYDataLogger get_dataLogger (): TYDataLogger
vb	function get_dataLogger () As YDataLogger
cs	YDataLogger get_dataLogger ()
java	YDataLogger get_dataLogger ()
uwp	async Task<YDataLogger> get_dataLogger ()
py	get_dataLogger ()
php	function get_dataLogger ()
ts	async get_dataLogger (): Promise<YDataLogger null>
es	async get_dataLogger ()
dnp	YDataLoggerProxy get_dataLogger ()
cp	YDataLoggerProxy* get_dataLogger ()

This method returns an object that can control global parameters of the data logger. The returned object should not be freed.

Returns :

an YDataLogger object, or null on error.

power→get_deliveredEnergyMeter()**YPower****power→deliveredEnergyMeter()**

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time, but only when positive.

js	function get_deliveredEnergyMeter ()
cpp	double get_deliveredEnergyMeter ()
m	-(double) deliveredEnergyMeter
pas	double get_deliveredEnergyMeter (): double
vb	function get_deliveredEnergyMeter () As Double
cs	double get_deliveredEnergyMeter ()
java	double get_deliveredEnergyMeter ()
uwp	async Task<double> get_deliveredEnergyMeter ()
py	get_deliveredEnergyMeter ()
php	function get_deliveredEnergyMeter ()
ts	async get_deliveredEnergyMeter (): Promise<number>
es	async get_deliveredEnergyMeter ()
dnp	double get_deliveredEnergyMeter ()
cp	double get_deliveredEnergyMeter ()
cmd	YPower target get_deliveredEnergyMeter

Note that this counter is reset at each start of the device.

Returns :

a floating point number corresponding to the energy counter, maintained by the wattmeter by integrating the power consumption over time, but only when positive

On failure, throws an exception or returns YPower . DELIVEREDENERGYMETER_INVALID.

power→get_errorMessage()**YPower****power→errorMessage()**

Returns the error message of the latest error with the electrical power sensor.

js	function get_errorMessage ()
cpp	string get_errorMessage ()
m	-(NSString*) errorMessage
pas	string get_errorMessage (): string
vb	function get_errorMessage () As String
cs	string get_errorMessage ()
java	String get_errorMessage ()
py	get_errorMessage ()
php	function get_errorMessage ()
ts	get_errorMessage (): string
es	get_errorMessage ()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the electrical power sensor object

power→get_errorType()**YPower****power→errorType()**

Returns the numerical error code of the latest error with the electrical power sensor.

js	function get_errorType ()
cpp	YRETCODE get_errorType ()
m	-(YRETCODE) errorType
pas	YRETCODE get_errorType (): YRETCODE
vb	function get_errorType () As YRETCODE
cs	YRETCODE get_errorType ()
java	int get_errorType ()
py	get_errorType ()
php	function get_errorType ()
ts	get_errorType (): number
es	get_errorType ()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the electrical power sensor object

power→get_friendlyName()**YPower****power→friendlyName()**

Returns a global identifier of the electrical power sensor in the format `MODULE_NAME.FUNCTION_NAME`.

js	<code>function get_friendlyName()</code>
cpp	<code>string get_friendlyName()</code>
m	<code>-(NSString*) friendlyName</code>
cs	<code>string get_friendlyName()</code>
java	<code>String get_friendlyName()</code>
py	<code>get_friendlyName()</code>
php	<code>function get_friendlyName()</code>
ts	<code>async get_friendlyName(): Promise<string></code>
es	<code>async get_friendlyName()</code>
dnp	<code>string get_friendlyName()</code>
cp	<code>string get_friendlyName()</code>

The returned string uses the logical names of the module and of the electrical power sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the electrical power sensor (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the electrical power sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `YPower.FRIENDLYNAME_INVALID`.

power→get_functionDescriptor()**YPower****power→functionDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

js	function get_functionDescriptor ()
cpp	YFUN_DESCR get_functionDescriptor ()
m	-(YFUN_DESCR) functionDescriptor
pas	YFUN_DESCR get_functionDescriptor (): YFUN_DESCR
vb	function get_functionDescriptor () As YFUN_DESCR
cs	YFUN_DESCR get_functionDescriptor ()
java	String get_functionDescriptor ()
py	get_functionDescriptor ()
php	function get_functionDescriptor ()
ts	async get_functionDescriptor (): Promise<string>
es	async get_functionDescriptor ()

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y\$CLASSNAME\$.FUNCTIONDESCRIPTOR_INVALID.

power→get_functionId()**YPower****power→functionId()**

Returns the hardware identifier of the electrical power sensor, without reference to the module.

js	function get_functionId ()
cpp	string get_functionId ()
m	-(NSString*) functionId
vb	function get_functionId () As String
cs	string get_functionId ()
java	String get_functionId ()
py	get_functionId ()
php	function get_functionId ()
ts	async get_functionId (): Promise<string>
es	async get_functionId ()
dnp	string get_functionId ()
cp	string get_functionId ()

For example `re_lay1`

Returns :

a string that identifies the electrical power sensor (ex: `re_lay1`)

On failure, throws an exception or returns `YPower . FUNCTIONID_INVALID`.

power→**get_hardwareId()****YPower****power**→**hardwareId()**

Returns the unique hardware identifier of the electrical power sensor in the form `SERIAL.FUNCTIONID`.

js	function get_hardwareId ()
cpp	string get_hardwareId ()
m	-(NSString*) hardwareId
vb	function get_hardwareId () As String
cs	string get_hardwareId ()
java	String get_hardwareId ()
py	get_hardwareId ()
php	function get_hardwareId ()
ts	async get_hardwareId (): Promise<string>
es	async get_hardwareId ()
dnp	string get_hardwareId ()
cp	string get_hardwareId ()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the electrical power sensor (for example `RELAYL01-123456.relay1`).

Returns :

a string that uniquely identifies the electrical power sensor (ex: `RELAYL01-123456.relay1`)

On failure, throws an exception or returns `YPower.HARDWAREID_INVALID`.

power→get_highestValue()**YPower****power→highestValue()**

Returns the maximal value observed for the electrical power since the device was started.

js	function get_highestValue ()
cpp	double get_highestValue ()
m	-(double) highestValue
pas	double get_highestValue (): double
vb	function get_highestValue () As Double
cs	double get_highestValue ()
java	double get_highestValue ()
uwp	async Task<double> get_highestValue ()
py	get_highestValue ()
php	function get_highestValue ()
ts	async get_highestValue (): Promise<number>
es	async get_highestValue ()
dnp	double get_highestValue ()
cp	double get_highestValue ()
cmd	YPower target get_highestValue

Can be reset to an arbitrary value thanks to `set_highestValue()`.

Returns :

a floating point number corresponding to the maximal value observed for the electrical power since the device was started

On failure, throws an exception or returns `YPower .HIGHESTVALUE_INVALID`.

power→get_logFrequency()**YPower****power→logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

js	function get_logFrequency ()
cpp	string get_logFrequency ()
m	-(NSString*) logFrequency
pas	string get_logFrequency (): string
vb	function get_logFrequency () As String
cs	string get_logFrequency ()
java	String get_logFrequency ()
uwp	async Task<string> get_logFrequency ()
py	get_logFrequency ()
php	function get_logFrequency ()
ts	async get_logFrequency (): Promise<string>
es	async get_logFrequency ()
dnp	string get_logFrequency ()
cp	string get_logFrequency ()
cmd	YPower target get_logFrequency

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns YPower . LOGFREQUENCY_INVALID.

power→get_logicalName()**YPower****power→logicalName()**

Returns the logical name of the electrical power sensor.

js	function get_logicalName ()
cpp	string get_logicalName ()
m	-(NSString*) logicalName
pas	string get_logicalName (): string
vb	function get_logicalName () As String
cs	string get_logicalName ()
java	String get_logicalName ()
uwp	async Task<string> get_logicalName ()
py	get_logicalName ()
php	function get_logicalName ()
ts	async get_logicalName (): Promise<string>
es	async get_logicalName ()
dnp	string get_logicalName ()
cp	string get_logicalName ()
cmd	YPower target get_logicalName

Returns :

a string corresponding to the logical name of the electrical power sensor.

On failure, throws an exception or returns YPower . LOGICALNAME_INVALID.

power→get_lowestValue()**YPower****power→lowestValue()**

Returns the minimal value observed for the electrical power since the device was started.

js	function get_lowestValue ()
cpp	double get_lowestValue ()
m	-(double) lowestValue
pas	double get_lowestValue (): double
vb	function get_lowestValue () As Double
cs	double get_lowestValue ()
java	double get_lowestValue ()
uwp	async Task<double> get_lowestValue ()
py	get_lowestValue ()
php	function get_lowestValue ()
ts	async get_lowestValue (): Promise<number>
es	async get_lowestValue ()
dnp	double get_lowestValue ()
cp	double get_lowestValue ()
cmd	YPower target get_lowestValue

Can be reset to an arbitrary value thanks to `set_lowestValue()`.

Returns :

a floating point number corresponding to the minimal value observed for the electrical power since the device was started

On failure, throws an exception or returns YPower . LOWESTVALUE_INVALID.

power→get_meter()**YPower****power→meter()**

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time, but only when positive.

js	function get_meter ()
cpp	double get_meter ()
m	-(double) meter
pas	double get_meter (): double
vb	function get_meter () As Double
cs	double get_meter ()
java	double get_meter ()
uwp	async Task<double> get_meter ()
py	get_meter ()
php	function get_meter ()
ts	async get_meter (): Promise<number>
es	async get_meter ()
dnf	double get_meter ()
cp	double get_meter ()
cmd	YPower target get_meter

Note that this counter is reset at each start of the device.

Returns :

a floating point number corresponding to the energy counter, maintained by the wattmeter by integrating the power consumption over time, but only when positive

On failure, throws an exception or returns YPower . METER_INVALID.

power→get_meterTimer()**YPower****power→meterTimer()**

Returns the elapsed time since last energy counter reset, in seconds.

js	function get_meterTimer ()
cpp	int get_meterTimer ()
m	-(int) meterTimer
pas	LongInt get_meterTimer (): LongInt
vb	function get_meterTimer () As Integer
cs	int get_meterTimer ()
java	int get_meterTimer ()
uwp	async Task<int> get_meterTimer ()
py	get_meterTimer ()
php	function get_meterTimer ()
ts	async get_meterTimer (): Promise<number>
es	async get_meterTimer ()
dnp	int get_meterTimer ()
cp	int get_meterTimer ()
cmd	YPower target get_meterTimer

Returns :

an integer corresponding to the elapsed time since last energy counter reset, in seconds

On failure, throws an exception or returns YPower . METERTIMER_INVALID.

power→get_module()**YPower****power→module()**

Gets the YModule object for the device on which the function is located.

js	function get_module ()
cpp	YModule * get_module ()
m	-(YModule*) module
pas	TYModule get_module (): TYModule
vb	function get_module () As YModule
cs	YModule get_module ()
java	YModule get_module ()
py	get_module ()
php	function get_module ()
ts	async get_module (): Promise<YModule>
es	async get_module ()
dnp	YModuleProxy get_module ()
cp	YModuleProxy * get_module ()

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

power→get_module_async()**YPower****power→module_async()**

Gets the YModule object for the device on which the function is located (asynchronous version).

```
js function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as on-line.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking Firefox JavaScript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous JavaScript calls for more details.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

power→get_receivedEnergyMeter()**YPower****power→receivedEnergyMeter()**

Returns the energy counter, maintained by the wattmeter by integrating the power consumption over time, but only when negative.

js	function get_receivedEnergyMeter ()
cpp	double get_receivedEnergyMeter ()
m	-(double) receivedEnergyMeter
pas	double get_receivedEnergyMeter (): double
vb	function get_receivedEnergyMeter () As Double
cs	double get_receivedEnergyMeter ()
java	double get_receivedEnergyMeter ()
uwp	async Task<double> get_receivedEnergyMeter ()
py	get_receivedEnergyMeter ()
php	function get_receivedEnergyMeter ()
ts	async get_receivedEnergyMeter (): Promise<number>
es	async get_receivedEnergyMeter ()
dnp	double get_receivedEnergyMeter ()
cp	double get_receivedEnergyMeter ()
cmd	YPower target get_receivedEnergyMeter

Note that this counter is reset at each start of the device.

Returns :

a floating point number corresponding to the energy counter, maintained by the wattmeter by integrating the power consumption over time, but only when negative

On failure, throws an exception or returns YPower .RECEIVEDENERGYMETER_INVALID.

power→get_recordedData()**YPower****power→recordedData()**

Retrieves a `YDataSet` object holding historical data for this sensor, for a specified time interval.

js	<code>function get_recordedData(startTime, endTime)</code>
cpp	<code>YDataSet get_recordedData(double startTime, double endTime)</code>
m	<code>-(YDataSet*) recordedData : (double) startTime : (double) endTime</code>
pas	<code>TYDataSet get_recordedData(startTime: double, endTime: double): TYDataSet</code>
vb	<code>function get_recordedData(ByVal startTime As Double, ByVal endTime As Double) As YDataSet</code>
cs	<code>YDataSet get_recordedData(double startTime, double endTime)</code>
java	<code>YDataSet get_recordedData(double startTime, double endTime)</code>
uwp	<code>async Task<YDataSet> get_recordedData(double startTime, double endTime)</code>
py	<code>get_recordedData(startTime, endTime)</code>
php	<code>function get_recordedData(\$startTime, \$endTime)</code>
ts	<code>async get_recordedData(startTime: number, endTime: number): Promise<YDataSet></code>
es	<code>async get_recordedData(startTime, endTime)</code>
dnp	<code>YDataSetProxy get_recordedData(double startTime, double endTime)</code>
cp	<code>YDataSetProxy* get_recordedData(double startTime, double endTime)</code>
cmd	<code>YPower target get_recordedData startTime endTime</code>

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the `YDataSet` class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as `YDataSet` objects are not supported by firmwares older than version 13000.

Parameters :

- startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without initial limit.
- endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without ending limit.

Returns :

an instance of `YDataSet`, providing access to historical data. Past measures can be loaded progressively using methods from the `YDataSet` object.

power→get_reportFrequency()**YPower****power→reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

js	function get_reportFrequency ()
cpp	string get_reportFrequency ()
m	-(NSString*) reportFrequency
pas	string get_reportFrequency (): string
vb	function get_reportFrequency () As String
cs	string get_reportFrequency ()
java	String get_reportFrequency ()
uwp	async Task<string> get_reportFrequency ()
py	get_reportFrequency ()
php	function get_reportFrequency ()
ts	async get_reportFrequency (): Promise<string>
es	async get_reportFrequency ()
dnp	string get_reportFrequency ()
cp	string get_reportFrequency ()
cmd	YPower target get_reportFrequency

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns YPower . REPORTFREQUENCY_INVALID.

power→get_resolution()**YPower****power→resolution()**

Returns the resolution of the measured values.

js	function get_resolution ()
cpp	double get_resolution ()
m	-(double) resolution
pas	double get_resolution (): double
vb	function get_resolution () As Double
cs	double get_resolution ()
java	double get_resolution ()
uwp	async Task<double> get_resolution ()
py	get_resolution ()
php	function get_resolution ()
ts	async get_resolution (): Promise<number>
es	async get_resolution ()
dnp	double get_resolution ()
cp	double get_resolution ()
cmd	YPower target get_resolution

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `YPower.RESOLUTION_INVALID`.

power→get_sensorState()**YPower****power→sensorState()**

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

js	function get_sensorState ()
cpp	int get_sensorState ()
m	-(int) sensorState
pas	LongInt get_sensorState (): LongInt
vb	function get_sensorState () As Integer
cs	int get_sensorState ()
java	int get_sensorState ()
uwp	async Task<int> get_sensorState ()
py	get_sensorState ()
php	function get_sensorState ()
ts	async get_sensorState (): Promise<number>
es	async get_sensorState ()
dnf	int get_sensorState ()
cp	int get_sensorState ()
cmd	YPower target get_sensorState

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns YPower . **SENSORSTATE_INVALID**.

power→get_serialNumber()**power→serialNumber()**

Returns the serial number of the module, as set by the factory.

js	function get_serialNumber ()
cpp	string get_serialNumber ()
m	-(NSString*) serialNumber
pas	string get_serialNumber (): string
vb	function get_serialNumber () As String
cs	string get_serialNumber ()
java	String get_serialNumber ()
uwp	async Task<string> get_serialNumber ()
py	get_serialNumber ()
php	function get_serialNumber ()
ts	async get_serialNumber (): Promise<string>
es	async get_serialNumber ()
dnp	string get_serialNumber ()
cp	string get_serialNumber ()
cmd	YPower target get_serialNumber

Returns :

a string corresponding to the serial number of the module, as set by the factory.

On failure, throws an exception or returns YFunction.SERIALNUMBER_INVALID.

power→get_unit()**YPower****power→unit()**

Returns the measuring unit for the electrical power.

js	function get_unit ()
cpp	string get_unit ()
m	-(NSString*) unit
pas	string get_unit (): string
vb	function get_unit () As String
cs	string get_unit ()
java	String get_unit ()
uwp	async Task<string> get_unit ()
py	get_unit ()
php	function get_unit ()
ts	async get_unit (): Promise<string>
es	async get_unit ()
dnp	string get_unit ()
cp	string get_unit ()
cmd	YPower target get_unit

Returns :

a string corresponding to the measuring unit for the electrical power

On failure, throws an exception or returns YPower .UNIT_INVALID.

power→get_userData()**power→userData()**

Returns the value of the userData attribute, as previously stored using method set_userData.

js	function get_userData ()
cpp	void * get_userData ()
m	-(id) userData
pas	Tobject get_userData (): Tobject
vb	function get_userData () As Object
cs	object get_userData ()
java	Object get_userData ()
py	get_userData ()
php	function get_userData ()
ts	async get_userData (): Promise<object null>
es	async get_userData ()

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

power→isOnline()**YPower**

Checks if the electrical power sensor is currently reachable, without raising any error.

js	function isOnline ()
cpp	bool isOnline ()
m	-(BOOL) isOnline
pas	boolean isOnline (): boolean
vb	function isOnline () As Boolean
cs	bool isOnline ()
java	boolean isOnline ()
py	isOnline ()
php	function isOnline ()
ts	async isOnline (): Promise<boolean>
es	async isOnline ()
dnp	bool isOnline ()
cp	bool isOnline ()

If there is a cached value for the electrical power sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the electrical power sensor.

Returns :

`true` if the electrical power sensor can be reached, and `false` otherwise

power→isOnline_async()**YPower**

Checks if the electrical power sensor is currently reachable, without raising any error (asynchronous version).

```
js function isOnline_async( callback, context)
```

If there is a cached value for the electrical power sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

power→isReadOnly()**YPower**

Test if the function is readOnly.

cpp	<code>bool isReadOnly()</code>
m	<code>-(bool) isReadOnly</code>
pas	<code>boolean isReadOnly(): boolean</code>
vb	<code>function isReadOnly() As Boolean</code>
cs	<code>bool isReadOnly()</code>
java	<code>boolean isReadOnly()</code>
uwp	<code>async Task<bool> isReadOnly()</code>
py	<code>isReadOnly()</code>
php	<code>function isReadOnly()</code>
ts	<code>async isReadOnly(): Promise<boolean></code>
es	<code>async isReadOnly()</code>
dnp	<code>bool isReadOnly()</code>
cp	<code>bool isReadOnly()</code>
cmd	<code>YPower target isReadOnly</code>

Return `true` if the function is write protected or that the function is not available.

Returns :

`true` if the function is readOnly or not online.

power→isSensorReady()**YPower**

Checks if the sensor is currently able to provide an up-to-date measure.

```
cmd YPower target isSensorReady
```

Returns false if the device is unreachable, or if the sensor does not have a current measure to transmit. No exception is raised if there is an error while trying to contact the device hosting \$THEFUNCTION\$.

Returns :

`true` if the sensor can provide an up-to-date measure, and `false` otherwise

power→load()**YPower**

Preloads the electrical power sensor cache with a specified validity duration.

js	function load (msValidity)
cpp	YRETCODE load (int msValidity)
m	-(YRETCODE) load : (u64) msValidity
pas	YRETCODE load (msValidity : u64): YRETCODE
vb	function load (ByVal msValidity As Long) As YRETCODE
cs	YRETCODE load (ulong msValidity)
java	int load (long msValidity)
py	load (msValidity)
php	function load (\$ msValidity)
ts	async load (msValidity : number): Promise<number>
es	async load (msValidity)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI . SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

power→loadAttribute()**YPower**

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

js	function loadAttribute (attrName)
cpp	string loadAttribute (string attrName)
m	-(NSString*) loadAttribute : (NSString*) attrName
pas	string loadAttribute (attrName : string): string
vb	function loadAttribute (ByVal attrName As String) As String
cs	string loadAttribute (string attrName)
java	String loadAttribute (String attrName)
uwp	async Task<string> loadAttribute (string attrName)
py	loadAttribute (attrName)
php	function loadAttribute (\$ attrName)
ts	async loadAttribute (attrName : string): Promise<string>
es	async loadAttribute (attrName)
dnp	string loadAttribute (string attrName)
cp	string loadAttribute (string attrName)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

power→loadCalibrationPoints()**YPower**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```

js  function loadCalibrationPoints( rawValues, refValues)
cpp int loadCalibrationPoints( vector<double> rawValues,
                             vector<double> refValues)
m   -(int) loadCalibrationPoints : (NSMutableArray*) rawValues
                             : (NSMutableArray*) refValues
pas LongInt loadCalibrationPoints( var rawValues: TDoubleArray,
                             var refValues: TDoubleArray): LongInt
vb  procedure loadCalibrationPoints( ByVal rawValues As List(Of)
cs  int loadCalibrationPoints( List<double> rawValues,
                             List<double> refValues)
java int loadCalibrationPoints( ArrayList<Double> rawValues,
                             ArrayList<Double> refValues)
uwp async Task<int> loadCalibrationPoints( List<double> rawValues,
                             List<double> refValues)
py  loadCalibrationPoints( rawValues, refValues)
php function loadCalibrationPoints( &$rawValues, &$refValues)
ts  async loadCalibrationPoints( rawValues: number[], refValues: number[]): Promise<number>
es  async loadCalibrationPoints( rawValues, refValues)
cmd YPower target loadCalibrationPoints rawValues refValues

```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→load_async()**YPower**

Preloads the electrical power sensor cache with a specified validity duration (asynchronous version).

```
js function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking the JavaScript virtual machine.

Parameters :

- msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds
- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or YAPI . SUCCESS)
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

power→muteValueCallbacks()**YPower**

Disables the propagation of every new advertised value to the parent hub.

js	function muteValueCallbacks ()
cpp	int muteValueCallbacks ()
m	-(int) muteValueCallbacks
pas	LongInt muteValueCallbacks (): LongInt
vb	function muteValueCallbacks () As Integer
cs	int muteValueCallbacks ()
java	int muteValueCallbacks ()
uwp	async Task<int> muteValueCallbacks ()
py	muteValueCallbacks ()
php	function muteValueCallbacks ()
ts	async muteValueCallbacks (): Promise<number>
es	async muteValueCallbacks ()
dnp	int muteValueCallbacks ()
cp	int muteValueCallbacks ()
cmd	YPower target muteValueCallbacks

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

YAPI . SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

power→nextPower()**YPower**

Continues the enumeration of electrical power sensors started using `yFirstPower()`.

js	<code>function nextPower()</code>
cpp	<code>YPower * nextPower()</code>
m	<code>-(nullable YPower*) nextPower</code>
pas	<code>TYPower nextPower(): TYPower</code>
vb	<code>function nextPower() As YPower</code>
cs	<code>YPower nextPower()</code>
java	<code>YPower nextPower()</code>
uwp	<code>YPower nextPower()</code>
py	<code>nextPower()</code>
php	<code>function nextPower()</code>
ts	<code>nextPower(): YPower null</code>
es	<code>nextPower()</code>

Caution: You can't make any assumption about the returned electrical power sensors order. If you want to find a specific a electrical power sensor, use `Power.findPower()` and a `hardwareID` or a logical name.

Returns :

a pointer to a `YPower` object, corresponding to a electrical power sensor currently online, or a `null` pointer if there are no more electrical power sensors to enumerate.

power→registerTimedReportCallback()**YPower**

Registers the callback function that is invoked on every periodic timed notification.

js	function registerTimedReportCallback (callback)
cpp	int registerTimedReportCallback (YPowerTimedReportCallback callback)
m	-(int) registerTimedReportCallback : (YPowerTimedReportCallback _Nullable) callback
pas	LongInt registerTimedReportCallback (callback : TYPowerTimedReportCallback): LongInt
vb	function registerTimedReportCallback (ByVal callback As YPowerTimedReportCallback) As Integer
cs	int registerTimedReportCallback (TimedReportCallback callback)
java	int registerTimedReportCallback (TimedReportCallback callback)
uwp	async Task<int> registerTimedReportCallback (TimedReportCallback callback)
py	registerTimedReportCallback (callback)
php	function registerTimedReportCallback (\$ callback)
ts	async registerTimedReportCallback (callback : YPowerTimedReportCallback null): Promise<number>
es	async registerTimedReportCallback (callback)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

power→registerValueCallback()**YPower**

Registers the callback function that is invoked on every change of advertised value.

js	function registerValueCallback (callback)
cpp	int registerValueCallback (YPowerValueCallback callback)
m	-(int) registerValueCallback : (YPowerValueCallback _Nullable) callback
pas	LongInt registerValueCallback (callback : TYPowerValueCallback): LongInt
vb	function registerValueCallback (ByVal callback As YPowerValueCallback) As Integer
cs	int registerValueCallback (ValueCallback callback)
java	int registerValueCallback (UpdateCallback callback)
uwp	async Task<int> registerValueCallback (ValueCallback callback)
py	registerValueCallback (callback)
php	function registerValueCallback (\$callback)
ts	async registerValueCallback (callback : YPowerValueCallback null): Promise<number>
es	async registerValueCallback (callback)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

power→reset()**YPower**

Resets the energy counters.

js	function reset ()
cpp	int reset ()
m	-(int) reset
pas	LongInt reset (): LongInt
vb	function reset () As Integer
cs	int reset ()
java	int reset ()
uwp	async Task<int> reset ()
py	reset ()
php	function reset ()
ts	async reset (): Promise<number>
es	async reset ()
dnp	int reset ()
cp	int reset ()
cmd	YPower target reset

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_advMode()**YPower****power→setAdvMode()**

Changes the measuring mode used for the advertised value pushed to the parent hub.

js	function set_advMode (newval)
cpp	int set_advMode (Y_ADVMODE_enum newval)
m	-(int) setAdvMode : (Y_ADVMODE_enum) newval
pas	integer set_advMode (newval : Integer): integer
vb	function set_advMode (ByVal newval As Integer) As Integer
cs	int set_advMode (int newval)
java	int set_advMode (int newval)
uwp	async Task<int> set_advMode (int newval)
py	set_advMode (newval)
php	function set_advMode (\$newval)
ts	async set_advMode (newval : YSensor_AdvMode): Promise<number>
es	async set_advMode (newval)
dnp	int set_advMode (int newval)
cp	int set_advMode (int newval)
cmd	YPower target set_advMode newval

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a value among `YPower.ADVMODE_IMMEDIATE`, `YPower.ADVMODE_PERIOD_AVG`, `YPower.ADVMODE_PERIOD_MIN` and `YPower.ADVMODE_PERIOD_MAX` corresponding to the measuring mode used for the advertised value pushed to the parent hub

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_highestValue()**YPower****power→setHighestValue()**

Changes the recorded maximal value observed.

js	function set_highestValue (newval)
cpp	int set_highestValue (double newval)
m	-(int) setHighestValue : (double) newval
pas	integer set_highestValue (newval : double): integer
vb	function set_highestValue (ByVal newval As Double) As Integer
cs	int set_highestValue (double newval)
java	int set_highestValue (double newval)
uwp	async Task<int> set_highestValue (double newval)
py	set_highestValue (newval)
php	function set_highestValue (\$newval)
ts	async set_highestValue (newval : number): Promise<number>
es	async set_highestValue (newval)
dnp	int set_highestValue (double newval)
cp	int set_highestValue (double newval)
cmd	YPower target set_highestValue newval

Can be used to reset the value returned by get_lowestValue().

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_logFrequency()**power→setLogFrequency()**

Changes the datalogger recording frequency for this function.

js	function set_logFrequency (newval)
cpp	int set_logFrequency (string newval)
m	-(int) setLogFrequency : (NSString*) newval
pas	integer set_logFrequency (newval : string): integer
vb	function set_logFrequency (ByVal newval As String) As Integer
cs	int set_logFrequency (string newval)
java	int set_logFrequency (String newval)
uwp	async Task<int> set_logFrequency (string newval)
py	set_logFrequency (newval)
php	function set_logFrequency (\$newval)
ts	async set_logFrequency (newval : string): Promise<number>
es	async set_logFrequency (newval)
dnp	int set_logFrequency (string newval)
cp	int set_logFrequency (string newval)
cmd	YPower target set_logFrequency newval

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF". Note that setting the datalogger recording frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_logicalName()**YPower****power→setLogicalName()**

Changes the logical name of the electrical power sensor.

js	function set_logicalName (newval)
cpp	int set_logicalName (string newval)
m	-(int) setLogicalName : (NSString*) newval
pas	integer set_logicalName (newval : string): integer
vb	function set_logicalName (ByVal newval As String) As Integer
cs	int set_logicalName (string newval)
java	int set_logicalName (String newval)
uwp	async Task<int> set_logicalName (string newval)
py	set_logicalName (newval)
php	function set_logicalName (\$ newval)
ts	async set_logicalName (newval : string): Promise<number>
es	async set_logicalName (newval)
dnp	int set_logicalName (string newval)
cp	int set_logicalName (string newval)
cmd	YPower target set_logicalName newval

You can use `yCheckLogicalName( )` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the electrical power sensor.

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_lowestValue()**power→setLowestValue()**

Changes the recorded minimal value observed.

js	function set_lowestValue (newval)
cpp	int set_lowestValue (double newval)
m	-(int) setLowestValue : (double) newval
pas	integer set_lowestValue (newval : double): integer
vb	function set_lowestValue (ByVal newval As Double) As Integer
cs	int set_lowestValue (double newval)
java	int set_lowestValue (double newval)
uwp	async Task<int> set_lowestValue (double newval)
py	set_lowestValue (newval)
php	function set_lowestValue (\$newval)
ts	async set_lowestValue (newval : number): Promise<number>
es	async set_lowestValue (newval)
dnp	int set_lowestValue (double newval)
cp	int set_lowestValue (double newval)
cmd	YPower target set_lowestValue newval

Can be used to reset the value returned by get_lowestValue().

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_reportFrequency()**YPower****power→setReportFrequency()**

Changes the timed value notification frequency for this function.

js	function set_reportFrequency (newval)
cpp	int set_reportFrequency (string newval)
m	-(int) setReportFrequency : (NSString*) newval
pas	integer set_reportFrequency (newval : string): integer
vb	function set_reportFrequency (ByVal newval As String) As Integer
cs	int set_reportFrequency (string newval)
java	int set_reportFrequency (String newval)
uwp	async Task<int> set_reportFrequency (string newval)
py	set_reportFrequency (newval)
php	function set_reportFrequency (\$ newval)
ts	async set_reportFrequency (newval : string): Promise<number>
es	async set_reportFrequency (newval)
dnp	int set_reportFrequency (string newval)
cp	int set_reportFrequency (string newval)
cmd	YPower target set_reportFrequency newval

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (e.g. "4/h"). To disable timed value notifications for this function, use the value "OFF". Note that setting the timed value notification frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_resolution()**power→setResolution()**

Changes the resolution of the measured physical values.

js	function set_resolution (newval)
cpp	int set_resolution (double newval)
m	-(int) setResolution : (double) newval
pas	integer set_resolution (newval : double): integer
vb	function set_resolution (ByVal newval As Double) As Integer
cs	int set_resolution (double newval)
java	int set_resolution (double newval)
uwp	async Task<int> set_resolution (double newval)
py	set_resolution (newval)
php	function set_resolution (\$newval)
ts	async set_resolution (newval : number): Promise<number>
es	async set_resolution (newval)
dnp	int set_resolution (double newval)
cp	int set_resolution (double newval)
cmd	YPower target set_resolution newval

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

power→set_userdata()**YPower****power→setUserData()**

Stores a user context provided as argument in the userData attribute of the function.

js	function set_userdata (data)
cpp	void set_userdata (void * data)
m	-(void) setUserData : (id) data
pas	set_userdata (data : Tobject)
vb	procedure set_userdata (ByVal data As Object)
cs	void set_userdata (object data)
java	void set_userdata (Object data)
py	set_userdata (data)
php	function set_userdata (\$ data)
ts	async set_userdata (data : object null): Promise<void>
es	async set_userdata (data)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

power→startDataLogger()**YPower**

Starts the data logger on the device.

js	function startDataLogger ()
cpp	int startDataLogger ()
m	-(int) startDataLogger
pas	LongInt startDataLogger (): LongInt
vb	function startDataLogger () As Integer
cs	int startDataLogger ()
java	int startDataLogger ()
uwp	async Task<int> startDataLogger ()
py	startDataLogger ()
php	function startDataLogger ()
ts	async startDataLogger (): Promise<number>
es	async startDataLogger ()
dnp	int startDataLogger ()
cp	int startDataLogger ()
cmd	YPower target startDataLogger

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :

YAPI . SUCCESS if the call succeeds.

power→stopDataLogger()**YPower**

Stops the datalogger on the device.

js	function stopDataLogger ()
cpp	int stopDataLogger ()
m	-(int) stopDataLogger
pas	LongInt stopDataLogger (): LongInt
vb	function stopDataLogger () As Integer
cs	int stopDataLogger ()
java	int stopDataLogger ()
uwp	async Task<int> stopDataLogger ()
py	stopDataLogger ()
php	function stopDataLogger ()
ts	async stopDataLogger (): Promise<number>
es	async stopDataLogger ()
dnp	int stopDataLogger ()
cp	int stopDataLogger ()
cmd	YPower target stopDataLogger

Returns :

YAPI . SUCCESS if the call succeeds.

power→unmuteValueCallbacks()**YPower**

Re-enables the propagation of every new advertised value to the parent hub.

js	function unmuteValueCallbacks ()
cpp	int unmuteValueCallbacks ()
m	-(int) unmuteValueCallbacks
pas	LongInt unmuteValueCallbacks (): LongInt
vb	function unmuteValueCallbacks () As Integer
cs	int unmuteValueCallbacks ()
java	int unmuteValueCallbacks ()
uwp	async Task<int> unmuteValueCallbacks ()
py	unmuteValueCallbacks ()
php	function unmuteValueCallbacks ()
ts	async unmuteValueCallbacks (): Promise<number>
es	async unmuteValueCallbacks ()
dnp	int unmuteValueCallbacks ()
cp	int unmuteValueCallbacks ()
cmd	YPower target unmuteValueCallbacks

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

YAPI . SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

power→wait_async()**YPower**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

js	<code>function wait_async(callback, context)</code>

ts	<code>wait_async(callback: Function, context: object)</code>

es	<code>wait_async(callback, context)</code>

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

24.4. Class YVoltage

Voltage sensor control interface, available for instance in the Yocto-Motor-DC, the Yocto-Volt or the Yocto-Watt

The YVoltage class allows you to read and configure Yoctopuce voltage sensors. It inherits from YSensor class the core functions to read measurements, to register callback functions, and to access the autonomous datalogger.

In order to use the functions described here, you should include:

es	in HTML: <code><script src='../lib/yocto_voltage.js'></script></code> in node.js: <code>require('yoctolib-es2017/yocto_voltage.js');</code>
js	<code><script type='text/javascript' src='yocto_voltage.js'></script></code>
cpp	<code>#include "yocto_voltage.h"</code>
m	<code>#import "yocto_voltage.h"</code>
pas	<code>uses yocto_voltage;</code>
vb	<code>yocto_voltage.vb</code>
cs	<code>yocto_voltage.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YVoltage;</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YVoltage;</code>
py	<code>from yocto_voltage import *</code>
php	<code>require_once('yocto_voltage.php');</code>
ts	in HTML: <code>import { YVoltage } from '../dist/esm/yocto_voltage.js';</code> in Node.js: <code>import { YVoltage } from 'yoctolib-cjs/yocto_voltage.js';</code>
dnp	<code>import YoctoProxyAPI.YVoltageProxy</code>
cp	<code>#include "yocto_voltage_proxy.h"</code>
vi	<code>YVoltage.vi</code>
ml	<code>import YoctoProxyAPI.YVoltageProxy</code>

Global functions

YVoltage.FindVoltage(func)

Retrieves a voltage sensor for a given identifier.

YVoltage.FindVoltageInContext(yctx, func)

Retrieves a voltage sensor for a given identifier in a YAPI context.

YVoltage.FirstVoltage()

Starts the enumeration of voltage sensors currently accessible.

YVoltage.FirstVoltageInContext(yctx)

Starts the enumeration of voltage sensors currently accessible.

YVoltage.GetSimilarFunctions()

Enumerates all functions of type Voltage available on the devices currently reachable by the library, and returns their unique hardware ID.

YVoltage properties

voltage→AdvMode [writable]

Measuring mode used for the advertised value pushed to the parent hub.

voltage→AdvertisedValue [read-only]

Short string representing the current state of the function.

voltage→Enabled [writable]

Activation state of this input.

voltage→FriendlyName [read-only]

Global identifier of the function in the format `MODULE_NAME . FUNCTION_NAME`.

voltage→FunctionId *[read-only]*

Hardware identifier of the sensor, without reference to the module.

voltage→HardwareId *[read-only]*

Unique hardware identifier of the function in the form `SERIAL . FUNCTIONID`.

voltage→IsOnline *[read-only]*

Checks if the function is currently reachable.

voltage→LogFrequency *[writable]*

Datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

voltage→LogicalName *[writable]*

Logical name of the function.

voltage→ReportFrequency *[writable]*

Timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

voltage→Resolution *[writable]*

Resolution of the measured values.

voltage→SerialNumber *[read-only]*

Serial number of the module, as set by the factory.

YVoltage methods

voltage→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

voltage→clearCache()

Invalidates the cache.

voltage→describe()

Returns a short text that describes unambiguously the instance of the voltage sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

voltage→get_advMode()

Returns the measuring mode used for the advertised value pushed to the parent hub.

voltage→get_advertisedValue()

Returns the current value of the voltage sensor (no more than 6 characters).

voltage→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number.

voltage→get_currentValue()

Returns the current value of the voltage, in Volt, as a floating point number.

voltage→get_dataLogger()

Returns the YData logger object of the device hosting the sensor.

voltage→get_enabled()

Returns the activation state of this input.

voltage→get_errorMessage()

Returns the error message of the latest error with the voltage sensor.

voltage→get_errorType()

Returns the numerical error code of the latest error with the voltage sensor.

voltage→get_friendlyName()

Returns a global identifier of the voltage sensor in the format `MODULE_NAME . FUNCTION_NAME`.

voltage→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

voltage→get_functionId()

Returns the hardware identifier of the voltage sensor, without reference to the module.

voltage→get_hardwareId()

Returns the unique hardware identifier of the voltage sensor in the form SERIAL . FUNCTIONID.

voltage→get_highestValue()

Returns the maximal value observed for the voltage since the device was started.

voltage→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

voltage→get_logicalName()

Returns the logical name of the voltage sensor.

voltage→get_lowestValue()

Returns the minimal value observed for the voltage since the device was started.

voltage→get_module()

Gets the YModule object for the device on which the function is located.

voltage→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

voltage→get_recordedData(startTime, endTime)

Retrieves a YDataSet object holding historical data for this sensor, for a specified time interval.

voltage→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

voltage→get_resolution()

Returns the resolution of the measured values.

voltage→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

voltage→get_serialNumber()

Returns the serial number of the module, as set by the factory.

voltage→get_unit()

Returns the measuring unit for the voltage.

voltage→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

voltage→isOnline()

Checks if the voltage sensor is currently reachable, without raising any error.

voltage→isOnline_async(callback, context)

Checks if the voltage sensor is currently reachable, without raising any error (asynchronous version).

voltage→isReadOnly()

Test if the function is readOnly.

voltage→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

voltage→load(msValidity)

Preloads the voltage sensor cache with a specified validity duration.

voltage→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.

voltage→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

voltage→load_async(msValidity, callback, context)

Preloads the voltage sensor cache with a specified validity duration (asynchronous version).

voltage→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

voltage→nextVoltage()

Continues the enumeration of voltage sensors started using `yFirstVoltage()`.

voltage→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

voltage→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

voltage→set_advMode(newval)

Changes the measuring mode used for the advertised value pushed to the parent hub.

voltage→set_enabled(newval)

Changes the activation state of this voltage input.

voltage→set_highestValue(newval)

Changes the recorded maximal value observed.

voltage→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

voltage→set_logicalName(newval)

Changes the logical name of the voltage sensor.

voltage→set_lowestValue(newval)

Changes the recorded minimal value observed.

voltage→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

voltage→set_resolution(newval)

Changes the resolution of the measured physical values.

voltage→set_userData(data)

Stores a user context provided as argument in the `userData` attribute of the function.

voltage→startDataLogger()

Starts the data logger on the device.

voltage→stopDataLogger()

Stops the datalogger on the device.

voltage→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

voltage→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YVoltage.FindVoltage()**YVoltage****YVoltage.FindVoltage()**

Retrieves a voltage sensor for a given identifier.

js	function yFindVoltage (func)
cpp	YVoltage* FindVoltage (string func)
m	+(YVoltage*) FindVoltage : (NSString*) func
pas	TYVoltage yFindVoltage (func : string): TYVoltage
vb	function FindVoltage (ByVal func As String) As YVoltage
cs	static YVoltage FindVoltage (string func)
java	static YVoltage FindVoltage (String func)
uwp	static YVoltage FindVoltage (string func)
py	FindVoltage (func)
php	function FindVoltage (\$func)
ts	static FindVoltage (func : string): YVoltage
es	static FindVoltage (func)
dnp	static YVoltageProxy FindVoltage (string func)
cp	static YVoltageProxy * FindVoltage (string func)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the voltage sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVoltage.isOnline()` to test if the voltage sensor is indeed online at a given time. In case of ambiguity when looking for a voltage sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

If a call to this object's `is_online()` method returns `FALSE` although you are certain that the matching device is plugged, make sure that you did call `registerHub()` at application initialization time.

Parameters :

func a string that uniquely characterizes the voltage sensor, for instance `MOTORCTL.voltage`.

Returns :

a `YVoltage` object allowing you to drive the voltage sensor.

YVoltage.FindVoltageInContext()**YVoltage****YVoltage.FindVoltageInContext()**

Retrieves a voltage sensor for a given identifier in a YAPI context.

java	static YVoltage FindVoltageInContext (YAPIContext yctx , String func)
uwp	static YVoltage FindVoltageInContext (YAPIContext yctx , string func)
ts	static FindVoltageInContext (yctx : YAPIContext, func : string): YVoltage
es	static FindVoltageInContext (yctx , func)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the voltage sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YVoltage.isOnline()` to test if the voltage sensor is indeed online at a given time. In case of ambiguity when looking for a voltage sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the voltage sensor, for instance `MOTORCTL.voltage`.

Returns :

a `YVoltage` object allowing you to drive the voltage sensor.

YVoltage.FirstVoltage()**YVoltage****YVoltage.FirstVoltage()**

Starts the enumeration of voltage sensors currently accessible.

js	function yFirstVoltage ()
cpp	YVoltage * FirstVoltage ()
m	+(YVoltage*) FirstVoltage
pas	TYVoltage yFirstVoltage (): TYVoltage
vb	function FirstVoltage () As YVoltage
cs	static YVoltage FirstVoltage ()
java	static YVoltage FirstVoltage ()
uwp	static YVoltage FirstVoltage ()
py	FirstVoltage ()
php	function FirstVoltage ()
ts	static FirstVoltage (): YVoltage null
es	static FirstVoltage ()

Use the method `YVoltage.nextVoltage()` to iterate on next voltage sensors.

Returns :

a pointer to a `YVoltage` object, corresponding to the first voltage sensor currently online, or a `null` pointer if there are none.

YVoltage.FirstVoltageInContext()**YVoltage****YVoltage.FirstVoltageInContext()**

Starts the enumeration of voltage sensors currently accessible.

java	static YVoltage FirstVoltageInContext (YAPIContext yctx)
uwp	static YVoltage FirstVoltageInContext (YAPIContext yctx)
ts	static FirstVoltageInContext (yctx : YAPIContext): YVoltage null
es	static FirstVoltageInContext (yctx)

Use the method `YVoltage.nextVoltage()` to iterate on next voltage sensors.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YVoltage` object, corresponding to the first voltage sensor currently online, or a `null` pointer if there are none.

YVoltage.GetSimilarFunctions()**YVoltage****YVoltage.GetSimilarFunctions()**

Enumerates all functions of type Voltage available on the devices currently reachable by the library, and returns their unique hardware ID.

dnp	<code>static new string[] GetSimilarFunctions()</code>

cp	<code>static vector<string> GetSimilarFunctions()</code>

Each of these IDs can be provided as argument to the method `YVoltage.FindVoltage` to obtain an object that can control the corresponding device.

Returns :

an array of strings, each string containing the unique hardwareId of a device function currently connected.

voltage→**AdvMode****YVoltage**

Measuring mode used for the advertised value pushed to the parent hub.

dnv **int AdvMode**

Writable. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

voltage→**AdvertisedValue****YVoltage**

Short string representing the current state of the function.

dnp

 string **AdvertisedValue**

voltage→Enabled**YVoltage**

Activation state of this input.

dnf **int Enabled**

Writable. Changes the activation state of this voltage input. When AC measurements are disabled, the device will always assume a DC signal, and vice-versa. When both AC and DC measurements are active, the device switches between AC and DC mode based on the relative amplitude of variations compared to the average value. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

voltage→FriendlyName**YVoltage**

Global identifier of the function in the format MODULE_NAME . FUNCTION_NAME.

dnv

 string **FriendlyName**

The returned string uses the logical names of the module and of the function if they are defined, otherwise the serial number of the module and the hardware identifier of the function (for example: MyCustomName.relay1)

voltage→**FunctionId****YVoltage**

Hardware identifier of the sensor, without reference to the module.

dnf [string](#) **FunctionId**

For example `re lay1`

voltage→HardwareId**YVoltage**

Unique hardware identifier of the function in the form SERIAL . FUNCTIONID.

dnp

 string **HardwareId**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example RELAYL01 - 123456 . r e l a y 1).

voltage→IsOnline**YVoltage**

Checks if the function is currently reachable.

dnf [bool IsOnline](#)

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the function.

voltage→LogFrequency**YVoltage**

Datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

dnv string **LogFrequency**

Writable. Changes the datalogger recording frequency for this function. The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF". Note that setting the datalogger recording frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

voltage→LogicalName**YVoltage**

Logical name of the function.

dnf `string LogicalName`

Writable. You can use `yCheckLogicalName( )` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

voltage→ReportFrequency**YVoltage**

Timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

dnf `string ReportFrequency`

Writable. Changes the timed value notification frequency for this function. The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (e.g. "4/h"). To disable timed value notifications for this function, use the value "OFF". Note that setting the timed value notification frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

voltage→Resolution**YVoltage**

Resolution of the measured values.

dnv **double Resolution**

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Writable. Changes the resolution of the measured physical values. The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

voltage→SerialNumber**YVoltage**

Serial number of the module, as set by the factory.

dnp string **SerialNumber**

voltage→calibrateFromPoints()**YVoltage**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

```

js function calibrateFromPoints( rawValues, refValues)
cpp int calibrateFromPoints( vector<double> rawValues,
                             vector<double> refValues)
m -(int) calibrateFromPoints : (NSMutableArray*) rawValues
   : (NSMutableArray*) refValues
pas LongInt calibrateFromPoints( rawValues: TDoubleArray,
                                refValues: TDoubleArray): LongInt
vb procedure calibrateFromPoints( ByVal rawValues As List(Of
cs int calibrateFromPoints( List<double> rawValues,
                             List<double> refValues)
java int calibrateFromPoints( ArrayList<Double> rawValues,
                              ArrayList<Double> refValues)
uwp async Task<int> calibrateFromPoints( List<double> rawValues,
                                         List<double> refValues)
py calibrateFromPoints( rawValues, refValues)
php function calibrateFromPoints( $rawValues, $refValues)
ts async calibrateFromPoints( rawValues: number[], refValues: number[]): Promise<number>
es async calibrateFromPoints( rawValues, refValues)
dnp int calibrateFromPoints( double[] rawValues,
                             double[] refValues)
cp int calibrateFromPoints( vector<double> rawValues,
                             vector<double> refValues)
cmd YVoltage target calibrateFromPoints rawValues refValues

```

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→clearCache()**YVoltage**

Invalidates the cache.

js	function clearCache ()
cpp	void clearCache ()
m	-(void) clearCache
pas	clearCache ()
vb	procedure clearCache ()
cs	void clearCache ()
java	void clearCache ()
py	clearCache ()
php	function clearCache ()
ts	async clearCache (): Promise<void>
es	async clearCache ()

Invalidates the cache of the voltage sensor attributes. Forces the next call to `get_xxx()` or `loadxxx()` to use values that come from the device.

voltage→describe()**YVoltage**

Returns a short text that describes unambiguously the instance of the voltage sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

js	function describe ()
cpp	string describe ()
m	-(NSString*) describe
pas	string describe (): string
vb	function describe () As String
cs	string describe ()
java	String describe ()
py	describe ()
php	function describe ()
ts	async describe (): Promise<string>
es	async describe ()

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYL01-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the voltage sensor (ex:
`Relay(MyCustomName.relay1)=RELAYL01-123456.relay1`)

voltage→get_advMode()**YVoltage****voltage→advMode()**

Returns the measuring mode used for the advertised value pushed to the parent hub.

js	function get_advMode ()
cpp	Y_ADVMODE_enum get_advMode ()
m	-(Y_ADVMODE_enum) advMode
pas	Integer get_advMode (): Integer
vb	function get_advMode () As Integer
cs	int get_advMode ()
java	int get_advMode ()
uwp	async Task<int> get_advMode ()
py	get_advMode ()
php	function get_advMode ()
ts	async get_advMode (): Promise<YSensor_AdvMode>
es	async get_advMode ()
dnp	int get_advMode ()
cp	int get_advMode ()
cmd	YVoltage target get_advMode

Returns :

a value among YVoltage.ADVMODE_IMMEDIATE, YVoltage.ADVMODE_PERIOD_AVG, YVoltage.ADVMODE_PERIOD_MIN and YVoltage.ADVMODE_PERIOD_MAX corresponding to the measuring mode used for the advertised value pushed to the parent hub

On failure, throws an exception or returns YVoltage.ADVMODE_INVALID.

voltage→**get_advertisedValue()****YVoltage****voltage**→**advertisedValue()**

Returns the current value of the voltage sensor (no more than 6 characters).

js	function get_advertisedValue ()
cpp	string get_advertisedValue ()
m	-(NSString*) advertisedValue
pas	string get_advertisedValue (): string
vb	function get_advertisedValue () As String
cs	string get_advertisedValue ()
java	String get_advertisedValue ()
uwp	async Task<string> get_advertisedValue ()
py	get_advertisedValue ()
php	function get_advertisedValue ()
ts	async get_advertisedValue (): Promise<string>
es	async get_advertisedValue ()
dnp	string get_advertisedValue ()
cp	string get_advertisedValue ()
cmd	YVoltage target get_advertisedValue

Returns :

a string corresponding to the current value of the voltage sensor (no more than 6 characters).

On failure, throws an exception or returns `YVoltage.AVERTISEDVALUE_INVALID`.

voltage→get_currentRawValue()**YVoltage****voltage→currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number.

js	function get_currentRawValue ()
cpp	double get_currentRawValue ()
m	-(double) currentRawValue
pas	double get_currentRawValue (): double
vb	function get_currentRawValue () As Double
cs	double get_currentRawValue ()
java	double get_currentRawValue ()
uwp	async Task<double> get_currentRawValue ()
py	get_currentRawValue ()
php	function get_currentRawValue ()
ts	async get_currentRawValue (): Promise<number>
es	async get_currentRawValue ()
dnp	double get_currentRawValue ()
cp	double get_currentRawValue ()
cmd	YVoltage target get_currentRawValue

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in Volt, as a floating point number

On failure, throws an exception or returns `YVoltage.CURRENTRAWVALUE_INVALID`.

voltage→**get_currentValue()****YVoltage****voltage**→**currentValue()**

Returns the current value of the voltage, in Volt, as a floating point number.

js	function get_currentValue ()
cpp	double get_currentValue ()
m	-(double) currentValue
pas	double get_currentValue (): double
vb	function get_currentValue () As Double
cs	double get_currentValue ()
java	double get_currentValue ()
uwp	async Task<double> get_currentValue ()
py	get_currentValue ()
php	function get_currentValue ()
ts	async get_currentValue (): Promise<number>
es	async get_currentValue ()
dnp	double get_currentValue ()
cp	double get_currentValue ()
cmd	YVoltage target get_currentValue

Note that a `get_currentValue()` call will **not** start a measure in the device, it will just return the last measure that occurred in the device. Indeed, internally, each Yoctopuce devices is continuously making measurements at a hardware specific frequency.

If continuously calling `get_currentValue()` leads you to performances issues, then you might consider to switch to callback programming model. Check the "advanced programming" chapter in in your device user manual for more information.

Returns :

a floating point number corresponding to the current value of the voltage, in Volt, as a floating point number

On failure, throws an exception or returns `YVoltage.CURRENTVALUE_INVALID`.

voltage→get_dataLogger()**YVoltage****voltage→dataLogger()**

Returns the YDataLogger object of the device hosting the sensor.

js	function get_dataLogger ()
cpp	YDataLogger* get_dataLogger ()
m	-(YDataLogger*) dataLogger
pas	TYDataLogger get_dataLogger (): TYDataLogger
vb	function get_dataLogger () As YDataLogger
cs	YDataLogger get_dataLogger ()
java	YDataLogger get_dataLogger ()
uwp	async Task<YDataLogger> get_dataLogger ()
py	get_dataLogger ()
php	function get_dataLogger ()
ts	async get_dataLogger (): Promise<YDataLogger null>
es	async get_dataLogger ()
dnp	YDataLoggerProxy get_dataLogger ()
cp	YDataLoggerProxy* get_dataLogger ()

This method returns an object that can control global parameters of the data logger. The returned object should not be freed.

Returns :

an YDataLogger object, or null on error.

voltage→**get_enabled()****YVoltage****voltage**→**enabled()**

Returns the activation state of this input.

js	function get_enabled ()
cpp	Y_ENABLED_enum get_enabled ()
m	-(Y_ENABLED_enum) enabled
pas	Integer get_enabled (): Integer
vb	function get_enabled () As Integer
cs	int get_enabled ()
java	int get_enabled ()
uwp	async Task<int> get_enabled ()
py	get_enabled ()
php	function get_enabled ()
ts	async get_enabled (): Promise<YVoltage_Enabled>
es	async get_enabled ()
dnp	int get_enabled ()
cp	int get_enabled ()
cmd	YVoltage target get_enabled

Returns :

either `YVoltage.ENABLED_FALSE` or `YVoltage.ENABLED_TRUE`, according to the activation state of this input

On failure, throws an exception or returns `YVoltage.ENABLED_INVALID`.

voltage→get_errorMessage()**YVoltage****voltage→errorMessage()**

Returns the error message of the latest error with the voltage sensor.

js	function get_errorMessage ()
cpp	string get_errorMessage ()
m	-(NSString*) errorMessage
pas	string get_errorMessage (): string
vb	function get_errorMessage () As String
cs	string get_errorMessage ()
java	String get_errorMessage ()
py	get_errorMessage ()
php	function get_errorMessage ()
ts	get_errorMessage (): string
es	get_errorMessage ()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the voltage sensor object

voltage→**get_errorType()****YVoltage****voltage**→**errorType()**

Returns the numerical error code of the latest error with the voltage sensor.

js	<code>function get_errorType()</code>
cpp	<code>YRETCODE get_errorType()</code>
m	<code>-(YRETCODE) errorType</code>
pas	<code>YRETCODE get_errorType(): YRETCODE</code>
vb	<code>function get_errorType() As YRETCODE</code>
cs	<code>YRETCODE get_errorType()</code>
java	<code>int get_errorType()</code>
py	<code>get_errorType()</code>
php	<code>function get_errorType()</code>
ts	<code>get_errorType(): number</code>
es	<code>get_errorType()</code>

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the voltage sensor object

voltage→get_friendlyName()**YVoltage****voltage→friendlyName()**

Returns a global identifier of the voltage sensor in the format `MODULE_NAME.FUNCTION_NAME`.

js	function get_friendlyName ()
cpp	string get_friendlyName ()
m	-(NSString*) friendlyName
cs	string get_friendlyName ()
java	String get_friendlyName ()
py	get_friendlyName ()
php	function get_friendlyName ()
ts	async get_friendlyName (): Promise<string>
es	async get_friendlyName ()
dnp	string get_friendlyName ()
cp	string get_friendlyName ()

The returned string uses the logical names of the module and of the voltage sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the voltage sensor (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the voltage sensor using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `YVoltage.FRIENDLYNAME_INVALID`.

voltage→**get_functionDescriptor()****YVoltage****voltage**→**functionDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

js	function get_functionDescriptor ()
cpp	YFUN_DESCR get_functionDescriptor ()
m	-(YFUN_DESCR) functionDescriptor
pas	YFUN_DESCR get_functionDescriptor (): YFUN_DESCR
vb	function get_functionDescriptor () As YFUN_DESCR
cs	YFUN_DESCR get_functionDescriptor ()
java	String get_functionDescriptor ()
py	get_functionDescriptor ()
php	function get_functionDescriptor ()
ts	async get_functionDescriptor (): Promise<string>
es	async get_functionDescriptor ()

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y\$CLASSNAME\$.FUNCTIONDESCRIPTOR_INVALID.

voltage→get_functionId()**YVoltage****voltage→functionId()**

Returns the hardware identifier of the voltage sensor, without reference to the module.

js	function get_functionId ()
cpp	string get_functionId ()
m	-(NSString*) functionId
vb	function get_functionId () As String
cs	string get_functionId ()
java	String get_functionId ()
py	get_functionId ()
php	function get_functionId ()
ts	async get_functionId (): Promise<string>
es	async get_functionId ()
dnp	string get_functionId ()
cp	string get_functionId ()

For example `relay1`

Returns :

a string that identifies the voltage sensor (ex: `relay1`)

On failure, throws an exception or returns `YVoltage.FUNCTIONID_INVALID`.

voltage→**get_hardwareId()****YVoltage****voltage**→**hardwareId()**

Returns the unique hardware identifier of the voltage sensor in the form `SERIAL . FUNCTIONID`.

js	function get_hardwareId ()
cpp	string get_hardwareId ()
m	-(NSString*) hardwareId
vb	function get_hardwareId () As String
cs	string get_hardwareId ()
java	String get_hardwareId ()
py	get_hardwareId ()
php	function get_hardwareId ()
ts	async get_hardwareId (): Promise<string>
es	async get_hardwareId ()
dnp	string get_hardwareId ()
cp	string get_hardwareId ()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the voltage sensor (for example `RELAYL01-123456 . relay1`).

Returns :

a string that uniquely identifies the voltage sensor (ex: `RELAYL01-123456 . relay1`)

On failure, throws an exception or returns `YVoltage.HARDWAREID_INVALID`.

voltage→**get_highestValue()****YVoltage****voltage**→**highestValue()**

Returns the maximal value observed for the voltage since the device was started.

js	function get_highestValue ()
cpp	double get_highestValue ()
m	-(double) highestValue
pas	double get_highestValue (): double
vb	function get_highestValue () As Double
cs	double get_highestValue ()
java	double get_highestValue ()
uwp	async Task<double> get_highestValue ()
py	get_highestValue ()
php	function get_highestValue ()
ts	async get_highestValue (): Promise<number>
es	async get_highestValue ()
dnp	double get_highestValue ()
cp	double get_highestValue ()
cmd	YVoltage target get_highestValue

Can be reset to an arbitrary value thanks to `set_highestValue()`.

Returns :

a floating point number corresponding to the maximal value observed for the voltage since the device was started

On failure, throws an exception or returns `YVoltage.HIGHESTVALUE_INVALID`.

voltage→get_logFrequency()**YVoltage****voltage→logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

js	function get_logFrequency ()
cpp	string get_logFrequency ()
m	-(NSString*) logFrequency
pas	string get_logFrequency (): string
vb	function get_logFrequency () As String
cs	string get_logFrequency ()
java	String get_logFrequency ()
uwp	async Task<string> get_logFrequency ()
py	get_logFrequency ()
php	function get_logFrequency ()
ts	async get_logFrequency (): Promise<string>
es	async get_logFrequency ()
dnp	string get_logFrequency ()
cp	string get_logFrequency ()
cmd	YVoltage target get_logFrequency

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns YVoltage.LOGFREQUENCY_INVALID.

voltage→get_logicalName()**YVoltage****voltage→logicalName()**

Returns the logical name of the voltage sensor.

js	function get_logicalName ()
cpp	string get_logicalName ()
m	-(NSString*) logicalName
pas	string get_logicalName (): string
vb	function get_logicalName () As String
cs	string get_logicalName ()
java	String get_logicalName ()
uwp	async Task<string> get_logicalName ()
py	get_logicalName ()
php	function get_logicalName ()
ts	async get_logicalName (): Promise<string>
es	async get_logicalName ()
dnp	string get_logicalName ()
cp	string get_logicalName ()
cmd	YVoltage target get_logicalName

Returns :

a string corresponding to the logical name of the voltage sensor.

On failure, throws an exception or returns `YVoltage.LOGICALNAME_INVALID`.

voltage→**get_lowestValue()****YVoltage****voltage**→**lowestValue()**

Returns the minimal value observed for the voltage since the device was started.

js	function get_lowestValue ()
cpp	double get_lowestValue ()
m	-(double) lowestValue
pas	double get_lowestValue (): double
vb	function get_lowestValue () As Double
cs	double get_lowestValue ()
java	double get_lowestValue ()
uwp	async Task<double> get_lowestValue ()
py	get_lowestValue ()
php	function get_lowestValue ()
ts	async get_lowestValue (): Promise<number>
es	async get_lowestValue ()
dnp	double get_lowestValue ()
cp	double get_lowestValue ()
cmd	YVoltage target get_lowestValue

Can be reset to an arbitrary value thanks to **set_lowestValue()**.

Returns :

a floating point number corresponding to the minimal value observed for the voltage since the device was started

On failure, throws an exception or returns **YVoltage.LOWESTVALUE_INVALID**.

voltage→get_module()**YVoltage****voltage→module()**

Gets the YModule object for the device on which the function is located.

js	function get_module ()
cpp	YModule * get_module ()
m	-(YModule*) module
pas	TYModule get_module (): TYModule
vb	function get_module () As YModule
cs	YModule get_module ()
java	YModule get_module ()
py	get_module ()
php	function get_module ()
ts	async get_module (): Promise<YModule>
es	async get_module ()
dnp	YModuleProxy get_module ()
cp	YModuleProxy * get_module ()

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

voltage→**get_module_async()****YVoltage****voltage**→**module_async()**

Gets the YModule object for the device on which the function is located (asynchronous version).

```
js function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as on-line.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking Firefox JavaScript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous JavaScript calls for more details.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

voltage→**get_recordedData()**

voltage→recordedData()

Retrieves a YDataSet object holding historical data for this sensor, for a specified time interval.

js	function get_recordedData (startTime , endTime)
c++	YDataSet get_recordedData (double startTime , double endTime)
m	-(YDataSet*) recordedData : (double) startTime : (double) endTime
pas	TYDataSet get_recordedData (startTime : double, endTime : double): TYDataSet
vb	function get_recordedData (ByVal startTime As Double, ByVal endTime As Double) As YDataSet
cs	YDataSet get_recordedData (double startTime , double endTime)
java	YDataSet get_recordedData (double startTime , double endTime)
uwp	async Task<YDataSet> get_recordedData (double startTime , double endTime)
py	get_recordedData (startTime , endTime)
php	function get_recordedData (\$startTime , \$endTime)
ts	async get_recordedData (startTime : number, endTime : number): Promise<YDataSet>
es	async get_recordedData (startTime , endTime)
dnp	YDataSetProxy get_recordedData (double startTime , double endTime)
cp	YDataSetProxy* get_recordedData (double startTime , double endTime)
cmd	YVoltage target get recordedData startTime endTime

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the `YDataSet` class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as YDataSet objects are not supported by firmwares older than version 13000.

Parameters :

startTime the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without initial limit.

endTime the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without ending limit.

Returns :

an instance of `YDataSet`, providing access to historical data. Past measures can be loaded progressively using methods from the `YDataSet` object.

voltage→**get_reportFrequency()****YVoltage****voltage**→**reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

js	function get_reportFrequency ()
cpp	string get_reportFrequency ()
m	-(NSString*) reportFrequency
pas	string get_reportFrequency (): string
vb	function get_reportFrequency () As String
cs	string get_reportFrequency ()
java	String get_reportFrequency ()
uwp	async Task<string> get_reportFrequency ()
py	get_reportFrequency ()
php	function get_reportFrequency ()
ts	async get_reportFrequency (): Promise<string>
es	async get_reportFrequency ()
dnp	string get_reportFrequency ()
cp	string get_reportFrequency ()
cmd	YVoltage target get_reportFrequency

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns `YVoltage.REPORTFREQUENCY_INVALID`.

voltage→get_resolution()**YVoltage****voltage→resolution()**

Returns the resolution of the measured values.

js	function get_resolution ()
cpp	double get_resolution ()
m	-(double) resolution
pas	double get_resolution (): double
vb	function get_resolution () As Double
cs	double get_resolution ()
java	double get_resolution ()
uwp	async Task<double> get_resolution ()
py	get_resolution ()
php	function get_resolution ()
ts	async get_resolution (): Promise<number>
es	async get_resolution ()
dnp	double get_resolution ()
cp	double get_resolution ()
cmd	YVoltage target get_resolution

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `YVoltage.RESOLUTION_INVALID`.

voltage→**get_sensorState()****YVoltage****voltage**→**sensorState()**

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

js	function get_sensorState ()
cpp	int get_sensorState ()
m	-(int) sensorState
pas	LongInt get_sensorState (): LongInt
vb	function get_sensorState () As Integer
cs	int get_sensorState ()
java	int get_sensorState ()
uwp	async Task<int> get_sensorState ()
py	get_sensorState ()
php	function get_sensorState ()
ts	async get_sensorState (): Promise<number>
es	async get_sensorState ()
dnp	int get_sensorState ()
cp	int get_sensorState ()
cmd	YVoltage target get_sensorState

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `YVoltage.SENSORSTATE_INVALID`.

voltage→get_serialNumber()**YVoltage****voltage→serialNumber()**

Returns the serial number of the module, as set by the factory.

js	function get_serialNumber ()
cpp	string get_serialNumber ()
m	-(NSString*) serialNumber
pas	string get_serialNumber (): string
vb	function get_serialNumber () As String
cs	string get_serialNumber ()
java	String get_serialNumber ()
uwp	async Task<string> get_serialNumber ()
py	get_serialNumber ()
php	function get_serialNumber ()
ts	async get_serialNumber (): Promise<string>
es	async get_serialNumber ()
dnp	string get_serialNumber ()
cp	string get_serialNumber ()
cmd	YVoltage target get_serialNumber

Returns :

a string corresponding to the serial number of the module, as set by the factory.

On failure, throws an exception or returns YFunction.SERIALNUMBER_INVALID.

voltage→**get_unit()****YVoltage****voltage**→**unit()**

Returns the measuring unit for the voltage.

js	function get_unit ()
cpp	string get_unit ()
m	-(NSString*) unit
pas	string get_unit (): string
vb	function get_unit () As String
cs	string get_unit ()
java	String get_unit ()
uwp	async Task<string> get_unit ()
py	get_unit ()
php	function get_unit ()
ts	async get_unit (): Promise<string>
es	async get_unit ()
dnp	string get_unit ()
cp	string get_unit ()
cmd	YVoltage target get_unit

Returns :

a string corresponding to the measuring unit for the voltage

On failure, throws an exception or returns YVoltage.UNIT_INVALID.

voltage→get_userData()**YVoltage****voltage→userData()**

Returns the value of the userData attribute, as previously stored using method set_userData.

js	function get_userData ()
cpp	void * get_userData ()
m	-(id) userData
pas	Tobject get_userData (): Tobject
vb	function get_userData () As Object
cs	object get_userData ()
java	Object get_userData ()
py	get_userData ()
php	function get_userData ()
ts	async get_userData (): Promise<object null>
es	async get_userData ()

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

voltage→isOnline()**YVoltage**

Checks if the voltage sensor is currently reachable, without raising any error.

js	<code>function isOnline()</code>
cpp	<code>bool isOnline()</code>
m	<code>-(BOOL) isOnline</code>
pas	<code>boolean isOnline(): boolean</code>
vb	<code>function isOnline() As Boolean</code>
cs	<code>bool isOnline()</code>
java	<code>boolean isOnline()</code>
py	<code>isOnline()</code>
php	<code>function isOnline()</code>
ts	<code>async isOnline(): Promise<boolean></code>
es	<code>async isOnline()</code>
dnp	<code>bool isOnline()</code>
cp	<code>bool isOnline()</code>

If there is a cached value for the voltage sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the voltage sensor.

Returns :

`true` if the voltage sensor can be reached, and `false` otherwise

voltage→isOnline_async()**YVoltage**

Checks if the voltage sensor is currently reachable, without raising any error (asynchronous version).

```
js function isOnline_async( callback, context)
```

If there is a cached value for the voltage sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

voltage→isReadOnly()**YVoltage**

Test if the function is readOnly.

cpp	<code>bool isReadOnly()</code>
m	<code>-(bool) isReadOnly</code>
pas	<code>boolean isReadOnly(): boolean</code>
vb	<code>function isReadOnly() As Boolean</code>
cs	<code>bool isReadOnly()</code>
java	<code>boolean isReadOnly()</code>
uwp	<code>async Task<bool> isReadOnly()</code>
py	<code>isReadOnly()</code>
php	<code>function isReadOnly()</code>
ts	<code>async isReadOnly(): Promise<boolean></code>
es	<code>async isReadOnly()</code>
dnp	<code>bool isReadOnly()</code>
cp	<code>bool isReadOnly()</code>
cmd	<code>YVoltage target isReadOnly</code>

Return `true` if the function is write protected or that the function is not available.

Returns :

`true` if the function is readOnly or not online.

voltage→isSensorReady()**YVoltage**

Checks if the sensor is currently able to provide an up-to-date measure.

```
cmd YVoltage target isSensorReady
```

Returns false if the device is unreachable, or if the sensor does not have a current measure to transmit. No exception is raised if there is an error while trying to contact the device hosting \$THEFUNCTION\$.

Returns :

`true` if the sensor can provide an up-to-date measure, and `false` otherwise

voltage→load()**YVoltage**

Preloads the voltage sensor cache with a specified validity duration.

js	function load (msValidity)
cpp	YRETCODE load (int msValidity)
m	-(YRETCODE) load : (u64) msValidity
pas	YRETCODE load (msValidity : u64): YRETCODE
vb	function load (ByVal msValidity As Long) As YRETCODE
cs	YRETCODE load (ulong msValidity)
java	int load (long msValidity)
py	load (msValidity)
php	function load (\$msValidity)
ts	async load (msValidity : number): Promise<number>
es	async load (msValidity)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI . SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→loadAttribute()**YVoltage**

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

js	function loadAttribute (attrName)
cpp	string loadAttribute (string attrName)
m	-(NSString*) loadAttribute : (NSString*) attrName
pas	string loadAttribute (attrName : string): string
vb	function loadAttribute (ByVal attrName As String) As String
cs	string loadAttribute (string attrName)
java	String loadAttribute (String attrName)
uwp	async Task<string> loadAttribute (string attrName)
py	loadAttribute (attrName)
php	function loadAttribute (\$ attrName)
ts	async loadAttribute (attrName : string): Promise<string>
es	async loadAttribute (attrName)
dnp	string loadAttribute (string attrName)
cp	string loadAttribute (string attrName)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

voltage→loadCalibrationPoints()**YVoltage**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```

js  function loadCalibrationPoints( rawValues, refValues)
cpp int loadCalibrationPoints( vector<double> rawValues,
                               vector<double> refValues)
m   -(int) loadCalibrationPoints : (NSMutableArray*) rawValues
                               : (NSMutableArray*) refValues
pas LongInt loadCalibrationPoints( var rawValues: TDoubleArray,
                                   var refValues: TDoubleArray): LongInt
vb  procedure loadCalibrationPoints( ByVal rawValues As List(Of)
cs  int loadCalibrationPoints( List<double> rawValues,
                               List<double> refValues)
java int loadCalibrationPoints( ArrayList<Double> rawValues,
                               ArrayList<Double> refValues)
uwp async Task<int> loadCalibrationPoints( List<double> rawValues,
                                           List<double> refValues)
py  loadCalibrationPoints( rawValues, refValues)
php function loadCalibrationPoints( &$rawValues, &$refValues)
ts  async loadCalibrationPoints( rawValues: number[], refValues: number[]): Promise<number>
es  async loadCalibrationPoints( rawValues, refValues)
cmd YVoltage target loadCalibrationPoints rawValues refValues

```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→load_async()**YVoltage**

Preloads the voltage sensor cache with a specified validity duration (asynchronous version).

```
js function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking the JavaScript virtual machine.

Parameters :

- msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds
- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or YAPI . SUCCESS)
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

voltage→muteValueCallbacks()**YVoltage**

Disables the propagation of every new advertised value to the parent hub.

js	function muteValueCallbacks ()
cpp	int muteValueCallbacks ()
m	-(int) muteValueCallbacks
pas	LongInt muteValueCallbacks (): LongInt
vb	function muteValueCallbacks () As Integer
cs	int muteValueCallbacks ()
java	int muteValueCallbacks ()
uwp	async Task<int> muteValueCallbacks ()
py	muteValueCallbacks ()
php	function muteValueCallbacks ()
ts	async muteValueCallbacks (): Promise<number>
es	async muteValueCallbacks ()
dnp	int muteValueCallbacks ()
cp	int muteValueCallbacks ()
cmd	YVoltage target muteValueCallbacks

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

YAPI . SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→nextVoltage()**YVoltage**

Continues the enumeration of voltage sensors started using `yFirstVoltage()`.

js	<code>function nextVoltage()</code>
cpp	<code>YVoltage * nextVoltage()</code>
m	<code>-(nullable YVoltage*) nextVoltage</code>
pas	<code>TYVoltage nextVoltage(): TYVoltage</code>
vb	<code>function nextVoltage() As YVoltage</code>
cs	<code>YVoltage nextVoltage()</code>
java	<code>YVoltage nextVoltage()</code>
uwp	<code>YVoltage nextVoltage()</code>
py	<code>nextVoltage()</code>
php	<code>function nextVoltage()</code>
ts	<code>nextVoltage(): YVoltage null</code>
es	<code>nextVoltage()</code>

Caution: You can't make any assumption about the returned voltage sensors order. If you want to find a specific a voltage sensor, use `Voltage.findVoltage()` and a `hardwareID` or a logical name.

Returns :

a pointer to a `YVoltage` object, corresponding to a voltage sensor currently online, or a `null` pointer if there are no more voltage sensors to enumerate.

voltage→registerTimedReportCallback()**YVoltage**

Registers the callback function that is invoked on every periodic timed notification.

js	function registerTimedReportCallback (callback)
cpp	int registerTimedReportCallback (YVoltageTimedReportCallback callback)
m	-(int) registerTimedReportCallback : (YVoltageTimedReportCallback _Nullable) callback
pas	LongInt registerTimedReportCallback (callback : TYVoltageTimedReportCallback): LongInt
vb	function registerTimedReportCallback (ByVal callback As YVoltageTimedReportCallback) As Integer
cs	int registerTimedReportCallback (TimedReportCallback callback)
java	int registerTimedReportCallback (TimedReportCallback callback)
uwp	async Task<int> registerTimedReportCallback (TimedReportCallback callback)
py	registerTimedReportCallback (callback)
php	function registerTimedReportCallback (\$ callback)
ts	async registerTimedReportCallback (callback : YVoltageTimedReportCallback null): Promise<number>
es	async registerTimedReportCallback (callback)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

voltage→registerValueCallback()**YVoltage**

Registers the callback function that is invoked on every change of advertised value.

js	function registerValueCallback (callback)
cpp	int registerValueCallback (YVoltageValueCallback callback)
m	-(int) registerValueCallback : (YVoltageValueCallback _Nullable) callback
pas	LongInt registerValueCallback (callback : TYVoltageValueCallback): LongInt
vb	function registerValueCallback (ByVal callback As YVoltageValueCallback) As Integer
cs	int registerValueCallback (ValueCallback callback)
java	int registerValueCallback (UpdateCallback callback)
uwp	async Task<int> registerValueCallback (ValueCallback callback)
py	registerValueCallback (callback)
php	function registerValueCallback (\$callback)
ts	async registerValueCallback (callback : YVoltageValueCallback null): Promise<number>
es	async registerValueCallback (callback)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

voltage→**set_advMode()****YVoltage****voltage**→**setAdvMode()**

Changes the measuring mode used for the advertised value pushed to the parent hub.

js	function set_advMode (newval)
cpp	int set_advMode (Y_ADVMODE_enum newval)
m	-(int) setAdvMode : (Y_ADVMODE_enum) newval
pas	integer set_advMode (newval : Integer): integer
vb	function set_advMode (ByVal newval As Integer) As Integer
cs	int set_advMode (int newval)
java	int set_advMode (int newval)
uwp	async Task<int> set_advMode (int newval)
py	set_advMode (newval)
php	function set_advMode (\$ newval)
ts	async set_advMode (newval : YSensor_AdvMode): Promise<number>
es	async set_advMode (newval)
dnp	int set_advMode (int newval)
cp	int set_advMode (int newval)
cmd	YVoltage target set_advMode newval

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a value among `YVoltage.ADVMODE_IMMEDIATE`, `YVoltage.ADVMODE_PERIOD_AVG`, `YVoltage.ADVMODE_PERIOD_MIN` and `YVoltage.ADVMODE_PERIOD_MAX` corresponding to the measuring mode used for the advertised value pushed to the parent hub

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→set_enabled()**YVoltage****voltage→setEnabled()**

Changes the activation state of this voltage input.

js	function set_enabled (newval)
cpp	int set_enabled (Y_ENABLED_enum newval)
m	-(int) setEnabled : (Y_ENABLED_enum) newval
pas	integer set_enabled (newval : Integer): integer
vb	function set_enabled (ByVal newval As Integer) As Integer
cs	int set_enabled (int newval)
java	int set_enabled (int newval)
uwp	async Task<int> set_enabled (int newval)
py	set_enabled (newval)
php	function set_enabled (\$ newval)
ts	async set_enabled (newval : YVoltage_Enabled): Promise<number>
es	async set_enabled (newval)
dnp	int set_enabled (int newval)
cp	int set_enabled (int newval)
cmd	YVoltage target set_enabled newval

When AC measurements are disabled, the device will always assume a DC signal, and vice-versa. When both AC and DC measurements are active, the device switches between AC and DC mode based on the relative amplitude of variations compared to the average value. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval either `YVoltage.ENABLED_FALSE` or `YVoltage.ENABLED_TRUE`, according to the activation state of this voltage input

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→**set_highestValue()****YVoltage****voltage**→**setHighestValue()**

Changes the recorded maximal value observed.

js	function set_highestValue (newval)
cpp	int set_highestValue (double newval)
m	-(int) setHighestValue : (double) newval
pas	integer set_highestValue (newval : double): integer
vb	function set_highestValue (ByVal newval As Double) As Integer
cs	int set_highestValue (double newval)
java	int set_highestValue (double newval)
uwp	async Task<int> set_highestValue (double newval)
py	set_highestValue (newval)
php	function set_highestValue (\$newval)
ts	async set_highestValue (newval : number): Promise<number>
es	async set_highestValue (newval)
dnp	int set_highestValue (double newval)
cp	int set_highestValue (double newval)
cmd	YVoltage target set_highestValue newval

Can be used to reset the value returned by `get_lowestValue()`.

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→set_logFrequency()**YVoltage****voltage→setLogFrequency()**

Changes the datalogger recording frequency for this function.

js	function set_logFrequency (newval)
cpp	int set_logFrequency (string newval)
m	-(int) setLogFrequency : (NSString*) newval
pas	integer set_logFrequency (newval : string): integer
vb	function set_logFrequency (ByVal newval As String) As Integer
cs	int set_logFrequency (string newval)
java	int set_logFrequency (String newval)
uwp	async Task<int> set_logFrequency (string newval)
py	set_logFrequency (newval)
php	function set_logFrequency (\$ newval)
ts	async set_logFrequency (newval : string): Promise<number>
es	async set_logFrequency (newval)
dnp	int set_logFrequency (string newval)
cp	int set_logFrequency (string newval)
cmd	YVoltage target set_logFrequency newval

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF". Note that setting the datalogger recording frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→**set_logicalName()****YVoltage****voltage**→**setLogicalName()**

Changes the logical name of the voltage sensor.

js	function set_logicalName (newval)
cpp	int set_logicalName (string newval)
m	-(int) setLogicalName : (NSString*) newval
pas	integer set_logicalName (newval : string): integer
vb	function set_logicalName (ByVal newval As String) As Integer
cs	int set_logicalName (string newval)
java	int set_logicalName (String newval)
uwp	async Task<int> set_logicalName (string newval)
py	set_logicalName (newval)
php	function set_logicalName (\$ newval)
ts	async set_logicalName (newval : string): Promise<number>
es	async set_logicalName (newval)
dnp	int set_logicalName (string newval)
cp	int set_logicalName (string newval)
cmd	YVoltage target set_logicalName newval

You can use `yCheckLogicalName( )` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the voltage sensor.

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→set_lowestValue()**YVoltage****voltage→setLowestValue()**

Changes the recorded minimal value observed.

js	function set_lowestValue (newval)
cpp	int set_lowestValue (double newval)
m	-(int) setLowestValue : (double) newval
pas	integer set_lowestValue (newval : double): integer
vb	function set_lowestValue (ByVal newval As Double) As Integer
cs	int set_lowestValue (double newval)
java	int set_lowestValue (double newval)
uwp	async Task<int> set_lowestValue (double newval)
py	set_lowestValue (newval)
php	function set_lowestValue (\$newval)
ts	async set_lowestValue (newval : number): Promise<number>
es	async set_lowestValue (newval)
dnp	int set_lowestValue (double newval)
cp	int set_lowestValue (double newval)
cmd	YVoltage target set_lowestValue newval

Can be used to reset the value returned by get_lowestValue().

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→set_reportFrequency()**YVoltage****voltage→setReportFrequency()**

Changes the timed value notification frequency for this function.

js	function set_reportFrequency (newval)
cpp	int set_reportFrequency (string newval)
m	-(int) setReportFrequency : (NSString*) newval
pas	integer set_reportFrequency (newval : string): integer
vb	function set_reportFrequency (ByVal newval As String) As Integer
cs	int set_reportFrequency (string newval)
java	int set_reportFrequency (String newval)
uwp	async Task<int> set_reportFrequency (string newval)
py	set_reportFrequency (newval)
php	function set_reportFrequency (\$ newval)
ts	async set_reportFrequency (newval : string): Promise<number>
es	async set_reportFrequency (newval)
dnp	int set_reportFrequency (string newval)
cp	int set_reportFrequency (string newval)
cmd	YVoltage target set_reportFrequency newval

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (e.g. "4/h"). To disable timed value notifications for this function, use the value "OFF". Note that setting the timed value notification frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→set_resolution()**YVoltage****voltage→setResolution()**

Changes the resolution of the measured physical values.

js	function set_resolution (newval)
cpp	int set_resolution (double newval)
m	-(int) setResolution : (double) newval
pas	integer set_resolution (newval : double): integer
vb	function set_resolution (ByVal newval As Double) As Integer
cs	int set_resolution (double newval)
java	int set_resolution (double newval)
uwp	async Task<int> set_resolution (double newval)
py	set_resolution (newval)
php	function set_resolution (\$newval)
ts	async set_resolution (newval : number): Promise<number>
es	async set_resolution (newval)
dnp	int set_resolution (double newval)
cp	int set_resolution (double newval)
cmd	YVoltage target set_resolution newval

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→**set_userData()****YVoltage****voltage**→**setUserData()**

Stores a user context provided as argument in the userData attribute of the function.

js	function set_userData (data)
cpp	void set_userData (void * data)
m	-(void) setUserData : (id) data
pas	set_userData (data : Tobject)
vb	procedure set_userData (ByVal data As Object)
cs	void set_userData (object data)
java	void set_userData (Object data)
py	set_userData (data)
php	function set_userData (\$ data)
ts	async set_userData (data : object null): Promise<void>
es	async set_userData (data)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

voltage→startDataLogger()**YVoltage**

Starts the data logger on the device.

js	function startDataLogger ()
cpp	int startDataLogger ()
m	-(int) startDataLogger
pas	LongInt startDataLogger (): LongInt
vb	function startDataLogger () As Integer
cs	int startDataLogger ()
java	int startDataLogger ()
uwp	async Task<int> startDataLogger ()
py	startDataLogger ()
php	function startDataLogger ()
ts	async startDataLogger (): Promise<number>
es	async startDataLogger ()
dnp	int startDataLogger ()
cp	int startDataLogger ()
cmd	YVoltage target startDataLogger

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :

YAPI . SUCCESS if the call succeeds.

voltage→stopDataLogger()**YVoltage**

Stops the datalogger on the device.

js	function stopDataLogger ()
cpp	int stopDataLogger ()
m	-(int) stopDataLogger
pas	LongInt stopDataLogger (): LongInt
vb	function stopDataLogger () As Integer
cs	int stopDataLogger ()
java	int stopDataLogger ()
uwp	async Task<int> stopDataLogger ()
py	stopDataLogger ()
php	function stopDataLogger ()
ts	async stopDataLogger (): Promise<number>
es	async stopDataLogger ()
dnp	int stopDataLogger ()
cp	int stopDataLogger ()
cmd	YVoltage target stopDataLogger

Returns :

YAPI . SUCCESS if the call succeeds.

voltage→unmuteValueCallbacks()**YVoltage**

Re-enables the propagation of every new advertised value to the parent hub.

js	function unmuteValueCallbacks ()
cpp	int unmuteValueCallbacks ()
m	-(int) unmuteValueCallbacks
pas	LongInt unmuteValueCallbacks (): LongInt
vb	function unmuteValueCallbacks () As Integer
cs	int unmuteValueCallbacks ()
java	int unmuteValueCallbacks ()
uwp	async Task<int> unmuteValueCallbacks ()
py	unmuteValueCallbacks ()
php	function unmuteValueCallbacks ()
ts	async unmuteValueCallbacks (): Promise<number>
es	async unmuteValueCallbacks ()
dnp	int unmuteValueCallbacks ()
cp	int unmuteValueCallbacks ()
cmd	YVoltage target unmuteValueCallbacks

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

YAPI . SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

voltage→wait_async()**YVoltage**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

js	<code>function wait_async(callback, context)</code>

ts	<code>wait_async(callback: Function, context: object)</code>

es	<code>wait_async(callback, context)</code>

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

24.5. Class YCurrent

Current sensor control interface, available for instance in the Yocto-Amp, the Yocto-Motor-DC or the Yocto-Watt

The YCurrent class allows you to read and configure Yoctopuce current sensors. It inherits from YSensor class the core functions to read measurements, to register callback functions, and to access the autonomous datalogger.

In order to use the functions described here, you should include:

es	in HTML: <code><script src='../lib/yocto_current.js'></script></code> in node.js: <code>require('yoctolib-es2017/yocto_current.js');</code>
js	<code><script type='text/javascript' src='yocto_current.js'></script></code>
cpp	<code>#include "yocto_current.h"</code>
m	<code>#import "yocto_current.h"</code>
pas	<code>uses yocto_current;</code>
vb	<code>yocto_current.vb</code>
cs	<code>yocto_current.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YCurrent;</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YCurrent;</code>
py	<code>from yocto_current import *</code>
php	<code>require_once('yocto_current.php');</code>
ts	in HTML: <code>import { YCurrent } from '../dist/esm/yocto_current.js';</code> in Node.js: <code>import { YCurrent } from 'yoctolib-cjs/yocto_current.js';</code>
dnf	<code>import YoctoProxyAPI.YCurrentProxy</code>
cp	<code>#include "yocto_current_proxy.h"</code>
vi	<code>YCurrent.vi</code>
ml	<code>import YoctoProxyAPI.YCurrentProxy</code>

Global functions

YCurrent.FindCurrent(func)

Retrieves a current sensor for a given identifier.

YCurrent.FindCurrentInContext(yctx, func)

Retrieves a current sensor for a given identifier in a YAPI context.

YCurrent.FirstCurrent()

Starts the enumeration of current sensors currently accessible.

YCurrent.FirstCurrentInContext(yctx)

Starts the enumeration of current sensors currently accessible.

YCurrent.GetSimilarFunctions()

Enumerates all functions of type Current available on the devices currently reachable by the library, and returns their unique hardware ID.

YCurrent properties

current→AdvMode [writable]

Measuring mode used for the advertised value pushed to the parent hub.

current→AdvertisedValue [read-only]

Short string representing the current state of the function.

current→Enabled [writable]

Activation state of this input.

current→FriendlyName [read-only]

Global identifier of the function in the format `MODULE_NAME . FUNCTION_NAME`.

current→FunctionId *[read-only]*

Hardware identifier of the sensor, without reference to the module.

current→HardwareId *[read-only]*

Unique hardware identifier of the function in the form `SERIAL . FUNCTIONID`.

current→IsOnline *[read-only]*

Checks if the function is currently reachable.

current→LogFrequency *[writable]*

Datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

current→LogicalName *[writable]*

Logical name of the function.

current→ReportFrequency *[writable]*

Timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

current→Resolution *[writable]*

Resolution of the measured values.

current→SerialNumber *[read-only]*

Serial number of the module, as set by the factory.

YCurrent methods

current→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

current→clearCache()

Invalidates the cache.

current→describe()

Returns a short text that describes unambiguously the instance of the current sensor in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

current→get_advMode()

Returns the measuring mode used for the advertised value pushed to the parent hub.

current→get_advertisedValue()

Returns the current value of the current sensor (no more than 6 characters).

current→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in mA, as a floating point number.

current→get_currentValue()

Returns the current value of the current, in mA, as a floating point number.

current→get_dataLogger()

Returns the YData logger object of the device hosting the sensor.

current→get_enabled()

Returns the activation state of this input.

current→get_errorMessage()

Returns the error message of the latest error with the current sensor.

current→get_errorType()

Returns the numerical error code of the latest error with the current sensor.

current→get_friendlyName()

Returns a global identifier of the current sensor in the format `MODULE_NAME . FUNCTION_NAME`.

current→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

current→get_functionId()

Returns the hardware identifier of the current sensor, without reference to the module.

current→get_hardwareId()

Returns the unique hardware identifier of the current sensor in the form SERIAL . FUNCTIONID.

current→get_highestValue()

Returns the maximal value observed for the current since the device was started.

current→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

current→get_logicalName()

Returns the logical name of the current sensor.

current→get_lowestValue()

Returns the minimal value observed for the current since the device was started.

current→get_module()

Gets the YModule object for the device on which the function is located.

current→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

current→get_recordedData(startTime, endTime)

Retrieves a YDataSet object holding historical data for this sensor, for a specified time interval.

current→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

current→get_resolution()

Returns the resolution of the measured values.

current→get_sensorState()

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

current→get_serialNumber()

Returns the serial number of the module, as set by the factory.

current→get_unit()

Returns the measuring unit for the current.

current→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

current→isOnline()

Checks if the current sensor is currently reachable, without raising any error.

current→isOnline_async(callback, context)

Checks if the current sensor is currently reachable, without raising any error (asynchronous version).

current→isReadOnly()

Test if the function is readOnly.

current→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

current→load(msValidity)

Preloads the current sensor cache with a specified validity duration.

current→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.

current→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

current→load_async(msValidity, callback, context)

Preloads the current sensor cache with a specified validity duration (asynchronous version).

current→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

current→nextCurrent()

Continues the enumeration of current sensors started using `yFirstCurrent()`.

current→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

current→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

current→set_advMode(newval)

Changes the measuring mode used for the advertised value pushed to the parent hub.

current→set_enabled(newval)

Changes the activation state of this voltage input.

current→set_highestValue(newval)

Changes the recorded maximal value observed.

current→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

current→set_logicalName(newval)

Changes the logical name of the current sensor.

current→set_lowestValue(newval)

Changes the recorded minimal value observed.

current→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

current→set_resolution(newval)

Changes the resolution of the measured physical values.

current→set_userData(data)

Stores a user context provided as argument in the `userData` attribute of the function.

current→startDataLogger()

Starts the data logger on the device.

current→stopDataLogger()

Stops the datalogger on the device.

current→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

current→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YCurrent.FindCurrent()**YCurrent****YCurrent.FindCurrent()**

Retrieves a current sensor for a given identifier.

js	function yFindCurrent (func)
cpp	YCurrent* FindCurrent (string func)
m	+(YCurrent*) FindCurrent : (NSString*) func
pas	TYCurrent yFindCurrent (func : string): TYCurrent
vb	function FindCurrent (ByVal func As String) As YCurrent
cs	static YCurrent FindCurrent (string func)
java	static YCurrent FindCurrent (String func)
uwp	static YCurrent FindCurrent (string func)
py	FindCurrent (func)
php	function FindCurrent (\$func)
ts	static FindCurrent (func : string): YCurrent
es	static FindCurrent (func)
dnp	static YCurrentProxy FindCurrent (string func)
cp	static YCurrentProxy * FindCurrent (string func)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the current sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YCurrent.isOnline()` to test if the current sensor is indeed online at a given time. In case of ambiguity when looking for a current sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

If a call to this object's `is_online()` method returns `FALSE` although you are certain that the matching device is plugged, make sure that you did call `registerHub()` at application initialization time.

Parameters :

func a string that uniquely characterizes the current sensor, for instance `YAMPMK01.current1`.

Returns :

a `YCurrent` object allowing you to drive the current sensor.

YCurrent.FindCurrentInContext()**YCurrent****YCurrent.FindCurrentInContext()**

Retrieves a current sensor for a given identifier in a YAPI context.

java	static YCurrent FindCurrentInContext (YAPIContext yctx , String func)
uwp	static YCurrent FindCurrentInContext (YAPIContext yctx , string func)
ts	static FindCurrentInContext (yctx : YAPIContext, func : string): YCurrent
es	static FindCurrentInContext (yctx , func)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the current sensor is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YCurrent.isOnline()` to test if the current sensor is indeed online at a given time. In case of ambiguity when looking for a current sensor by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the current sensor, for instance `YAMPMK01.current1`.

Returns :

a YCurrent object allowing you to drive the current sensor.

YCurrent.FirstCurrent()**YCurrent****YCurrent.FirstCurrent()**

Starts the enumeration of current sensors currently accessible.

js	function yFirstCurrent ()
cpp	YCurrent * FirstCurrent ()
m	+(YCurrent*) FirstCurrent
pas	TYCurrent yFirstCurrent (): TYCurrent
vb	function FirstCurrent () As YCurrent
cs	static YCurrent FirstCurrent ()
java	static YCurrent FirstCurrent ()
uwp	static YCurrent FirstCurrent ()
py	FirstCurrent ()
php	function FirstCurrent ()
ts	static FirstCurrent (): YCurrent null
es	static FirstCurrent ()

Use the method `YCurrent.nextCurrent( )` to iterate on next current sensors.

Returns :

a pointer to a `YCurrent` object, corresponding to the first current sensor currently online, or a `null` pointer if there are none.

YCurrent.FirstCurrentInContext() YCurrent.FirstCurrentInContext()

YCurrent

Starts the enumeration of current sensors currently accessible.

java	static YCurrent FirstCurrentInContext (YAPIContext yctx)
uwp	static YCurrent FirstCurrentInContext (YAPIContext yctx)
ts	static FirstCurrentInContext (yctx : YAPIContext): YCurrent null
es	static FirstCurrentInContext (yctx)

Use the method `YCurrent.nextCurrent()` to iterate on next current sensors.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YCurrent` object, corresponding to the first current sensor currently online, or a `null` pointer if there are none.

YCurrent.GetSimilarFunctions()**YCurrent****YCurrent.GetSimilarFunctions()**

Enumerates all functions of type Current available on the devices currently reachable by the library, and returns their unique hardware ID.

dnp	<code>static new string[] GetSimilarFunctions()</code>

cp	<code>static vector<string> GetSimilarFunctions()</code>

Each of these IDs can be provided as argument to the method `YCurrent.FindCurrent` to obtain an object that can control the corresponding device.

Returns :

an array of strings, each string containing the unique hardwareId of a device function currently connected.

current→**AdvMode****YCurrent**

Measuring mode used for the advertised value pushed to the parent hub.

dnf [int AdvMode](#)

Writable. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

current→**AdvertisedValue****YCurrent**

Short string representing the current state of the function.

dnp

 string **AdvertisedValue**

current→**Enabled****YCurrent**

Activation state of this input.

dnf **int Enabled**

Writable. Changes the activation state of this voltage input. When AC measurements are disabled, the device will always assume a DC signal, and vice-versa. When both AC and DC measurements are active, the device switches between AC and DC mode based on the relative amplitude of variations compared to the average value. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

current→FriendlyName**YCurrent**

Global identifier of the function in the format `MODULE_NAME . FUNCTION_NAME`.

`dn` `string` **FriendlyName**

The returned string uses the logical names of the module and of the function if they are defined, otherwise the serial number of the module and the hardware identifier of the function (for example: `MyCustomName.relay1`)

current→**FunctionId****YCurrent**

Hardware identifier of the sensor, without reference to the module.

dnf [string](#) **FunctionId**

For example `re lay1`

current→**HardwareId****YCurrent**

Unique hardware identifier of the function in the form SERIAL . FUNCTIONID.

dnf string **HardwareId**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example RELAYL01 - 123456 . r e l a y 1).

current→IsOnline**YCurrent**

Checks if the function is currently reachable.

dnf [bool IsOnline](#)

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the function.

current→LogFrequency**YCurrent**

Datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

dnf

string LogFrequency

Writable. Changes the datalogger recording frequency for this function. The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF". Note that setting the datalogger recording frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

current→LogicalName**YCurrent**

Logical name of the function.

dnf `string LogicalName`

Writable. You can use `yCheckLogicalName( )` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

current→ReportFrequency**YCurrent**

Timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

dnf `string ReportFrequency`

Writable. Changes the timed value notification frequency for this function. The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (e.g. "4/h"). To disable timed value notifications for this function, use the value "OFF". Note that setting the timed value notification frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

current→Resolution**YCurrent**

Resolution of the measured values.

dnf [double Resolution](#)

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor. Remember to call the `saveToF\ash( )` method of the module if the modification must be kept.

Writable. Changes the resolution of the measured physical values. The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself. Remember to call the `saveToF\ash( )` method of the module if the modification must be kept.

current→**SerialNumber****YCurrent**

Serial number of the module, as set by the factory.

dnp string **SerialNumber**

current→calibrateFromPoints()**YCurrent**

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

js	function calibrateFromPoints (rawValues , refValues)
cpp	int calibrateFromPoints (vector<double> rawValues , vector<double> refValues)
m	-(int) calibrateFromPoints : (NSMutableArray*) rawValues : (NSMutableArray*) refValues
pas	LongInt calibrateFromPoints (rawValues : TDoubleArray, refValues : TDoubleArray): LongInt
vb	procedure calibrateFromPoints (ByVal rawValues As List(Of
cs	int calibrateFromPoints (List<double> rawValues , List<double> refValues)
java	int calibrateFromPoints (ArrayList<Double> rawValues , ArrayList<Double> refValues)
uwp	async Task<int> calibrateFromPoints (List<double> rawValues , List<double> refValues)
py	calibrateFromPoints (rawValues , refValues)
php	function calibrateFromPoints (\$ rawValues , \$ refValues)
ts	async calibrateFromPoints (rawValues : number[], refValues : number[]): Promise<number>
es	async calibrateFromPoints (rawValues , refValues)
dnp	int calibrateFromPoints (double[] rawValues , double[] refValues)
cp	int calibrateFromPoints (vector<double> rawValues , vector<double> refValues)
cmd	YCurrent target calibrateFromPoints rawValues refValues

It is possible to configure up to five correction points. Correction points must be provided in ascending order, and be in the range of the sensor. The device will automatically perform a linear interpolation of the error correction between specified points. Remember to call the `saveToFFlash()` method of the module if the modification must be kept.

For more information on advanced capabilities to refine the calibration of sensors, please contact support@yoctopuce.com.

Parameters :

rawValues array of floating point numbers, corresponding to the raw values returned by the sensor for the correction points.

refValues array of floating point numbers, corresponding to the corrected values for the correction points.

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

current→clearCache()**YCurrent**

Invalidates the cache.

js	function clearCache ()
cpp	void clearCache ()
m	-(void) clearCache
pas	clearCache ()
vb	procedure clearCache ()
cs	void clearCache ()
java	void clearCache ()
py	clearCache ()
php	function clearCache ()
ts	async clearCache (): Promise<void>
es	async clearCache ()

Invalidates the cache of the current sensor attributes. Forces the next call to `get_xxx()` or `loadxxx()` to use values that come from the device.

current→describe()**YCurrent**

Returns a short text that describes unambiguously the instance of the current sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

js	function describe ()
cpp	string describe ()
m	-(NSString*) describe
pas	string describe (): string
vb	function describe () As String
cs	string describe ()
java	String describe ()
py	describe ()
php	function describe ()
ts	async describe (): Promise<string>
es	async describe ()

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns `Relay(MyCustomName.relay1)=RELAYL01-123456.relay1` if the module is already connected or `Relay(BadCustomName.relay1)=unresolved` if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the current sensor (ex:
`Relay(MyCustomName.relay1)=RELAYL01-123456.relay1`)

current→get_advMode()**YCurrent****current→advMode()**

Returns the measuring mode used for the advertised value pushed to the parent hub.

js	function get_advMode ()
cpp	Y_ADVMODE_enum get_advMode ()
m	-(Y_ADVMODE_enum) advMode
pas	Integer get_advMode (): Integer
vb	function get_advMode () As Integer
cs	int get_advMode ()
java	int get_advMode ()
uwp	async Task<int> get_advMode ()
py	get_advMode ()
php	function get_advMode ()
ts	async get_advMode (): Promise<YSensor_AdvMode>
es	async get_advMode ()
dnp	int get_advMode ()
cp	int get_advMode ()
cmd	YCurrent target get_advMode

Returns :

a value among YCurrent.ADVMODE_IMMEDIATE, YCurrent.ADVMODE_PERIOD_AVG, YCurrent.ADVMODE_PERIOD_MIN and YCurrent.ADVMODE_PERIOD_MAX corresponding to the measuring mode used for the advertised value pushed to the parent hub

On failure, throws an exception or returns YCurrent.ADVMODE_INVALID.

current→**get_advertisedValue()****YCurrent****current**→**advertisedValue()**

Returns the current value of the current sensor (no more than 6 characters).

js	function get_advertisedValue ()
cpp	string get_advertisedValue ()
m	-(NSString*) advertisedValue
pas	string get_advertisedValue (): string
vb	function get_advertisedValue () As String
cs	string get_advertisedValue ()
java	String get_advertisedValue ()
uwp	async Task<string> get_advertisedValue ()
py	get_advertisedValue ()
php	function get_advertisedValue ()
ts	async get_advertisedValue (): Promise<string>
es	async get_advertisedValue ()
dnp	string get_advertisedValue ()
cp	string get_advertisedValue ()
cmd	YCurrent target get_advertisedValue

Returns :

a string corresponding to the current value of the current sensor (no more than 6 characters).

On failure, throws an exception or returns **YCurrent.AVERTISEDVALUE_INVALID**.

current→**get_currentRawValue()****YCurrent****current**→**currentRawValue()**

Returns the uncalibrated, unrounded raw value returned by the sensor, in mA, as a floating point number.

js	function get_currentRawValue ()
cpp	double get_currentRawValue ()
m	-(double) currentRawValue
pas	double get_currentRawValue (): double
vb	function get_currentRawValue () As Double
cs	double get_currentRawValue ()
java	double get_currentRawValue ()
uwp	async Task<double> get_currentRawValue ()
py	get_currentRawValue ()
php	function get_currentRawValue ()
ts	async get_currentRawValue (): Promise<number>
es	async get_currentRawValue ()
dnp	double get_currentRawValue ()
cp	double get_currentRawValue ()
cmd	YCurrent target get_currentRawValue

Returns :

a floating point number corresponding to the uncalibrated, unrounded raw value returned by the sensor, in mA, as a floating point number

On failure, throws an exception or returns `YCurrent.CURRENTRAWVALUE_INVALID`.

current→**get_currentValue()****YCurrent****current**→**currentValue()**

Returns the current value of the current, in mA, as a floating point number.

js	function get_currentValue ()
cpp	double get_currentValue ()
m	-(double) currentValue
pas	double get_currentValue (): double
vb	function get_currentValue () As Double
cs	double get_currentValue ()
java	double get_currentValue ()
uwp	async Task<double> get_currentValue ()
py	get_currentValue ()
php	function get_currentValue ()
ts	async get_currentValue (): Promise<number>
es	async get_currentValue ()
dnp	double get_currentValue ()
cp	double get_currentValue ()
cmd	YCurrent target get_currentValue

Note that a `get_currentValue()` call will **not** start a measure in the device, it will just return the last measure that occurred in the device. Indeed, internally, each Yoctopuce devices is continuously making measurements at a hardware specific frequency.

If continuously calling `get_currentValue()` leads you to performances issues, then you might consider to switch to callback programming model. Check the "advanced programming" chapter in in your device user manual for more information.

Returns :

a floating point number corresponding to the current value of the current, in mA, as a floating point number

On failure, throws an exception or returns `YCurrent.CURRENTVALUE_INVALID`.

current→get_dataLogger()**YCurrent****current→dataLogger()**

Returns the YDataLogger object of the device hosting the sensor.

js	function get_dataLogger ()
cpp	YDataLogger* get_dataLogger ()
m	-(YDataLogger*) dataLogger
pas	TYDataLogger get_dataLogger (): TYDataLogger
vb	function get_dataLogger () As YDataLogger
cs	YDataLogger get_dataLogger ()
java	YDataLogger get_dataLogger ()
uwp	async Task<YDataLogger> get_dataLogger ()
py	get_dataLogger ()
php	function get_dataLogger ()
ts	async get_dataLogger (): Promise<YDataLogger null>
es	async get_dataLogger ()
dnp	YDataLoggerProxy get_dataLogger ()
cp	YDataLoggerProxy* get_dataLogger ()

This method returns an object that can control global parameters of the data logger. The returned object should not be freed.

Returns :

an YDataLogger object, or null on error.

current→get_enabled()**YCurrent****current→enabled()**

Returns the activation state of this input.

js	function get_enabled ()
cpp	Y_ENABLED_enum get_enabled ()
m	-(Y_ENABLED_enum) enabled
pas	Integer get_enabled (): Integer
vb	function get_enabled () As Integer
cs	int get_enabled ()
java	int get_enabled ()
uwp	async Task<int> get_enabled ()
py	get_enabled ()
php	function get_enabled ()
ts	async get_enabled (): Promise<YCurrent_Enabled>
es	async get_enabled ()
dnp	int get_enabled ()
cp	int get_enabled ()
cmd	YCurrent target get_enabled

Returns :

either `YCurrent.ENABLED_FALSE` or `YCurrent.ENABLED_TRUE`, according to the activation state of this input

On failure, throws an exception or returns `YCurrent.ENABLED_INVALID`.

current→get_errorMessage()**YCurrent****current→errorMessage()**

Returns the error message of the latest error with the current sensor.

js	function get_errorMessage ()
cpp	string get_errorMessage ()
m	-(NSString*) errorMessage
pas	string get_errorMessage (): string
vb	function get_errorMessage () As String
cs	string get_errorMessage ()
java	String get_errorMessage ()
py	get_errorMessage ()
php	function get_errorMessage ()
ts	get_errorMessage (): string
es	get_errorMessage ()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the current sensor object

current→get_errorType()**YCurrent****current→errorType()**

Returns the numerical error code of the latest error with the current sensor.

js	function get_errorType ()
cpp	YRETCODE get_errorType ()
m	-(YRETCODE) errorType
pas	YRETCODE get_errorType (): YRETCODE
vb	function get_errorType () As YRETCODE
cs	YRETCODE get_errorType ()
java	int get_errorType ()
py	get_errorType ()
php	function get_errorType ()
ts	get_errorType (): number
es	get_errorType ()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the current sensor object

current→get_friendlyName()**YCurrent****current→friendlyName()**

Returns a global identifier of the current sensor in the format `MODULE_NAME . FUNCTION_NAME`.

js	function get_friendlyName ()
cpp	string get_friendlyName ()
m	-(NSString*) friendlyName
cs	string get_friendlyName ()
java	String get_friendlyName ()
py	get_friendlyName ()
php	function get_friendlyName ()
ts	async get_friendlyName (): Promise<string>
es	async get_friendlyName ()
dnp	string get_friendlyName ()
cp	string get_friendlyName ()

The returned string uses the logical names of the module and of the current sensor if they are defined, otherwise the serial number of the module and the hardware identifier of the current sensor (for example: `MyCustomName . relay1`)

Returns :

a string that uniquely identifies the current sensor using logical names (ex: `MyCustomName . relay1`)

On failure, throws an exception or returns `YCurrent . FRIENDLYNAME_INVALID`.

current→**get_functionDescriptor()****YCurrent****current**→**functionDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

js	function get_functionDescriptor ()
cpp	YFUN_DESCR get_functionDescriptor ()
m	-(YFUN_DESCR) functionDescriptor
pas	YFUN_DESCR get_functionDescriptor (): YFUN_DESCR
vb	function get_functionDescriptor () As YFUN_DESCR
cs	YFUN_DESCR get_functionDescriptor ()
java	String get_functionDescriptor ()
py	get_functionDescriptor ()
php	function get_functionDescriptor ()
ts	async get_functionDescriptor (): Promise<string>
es	async get_functionDescriptor ()

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y\$CLASSNAME\$.FUNCTIONDESCRIPTOR_INVALID.

current→**get_functionId()****YCurrent****current**→**functionId()**

Returns the hardware identifier of the current sensor, without reference to the module.

js	function get_functionId ()
cpp	string get_functionId ()
m	-(NSString*) functionId
vb	function get_functionId () As String
cs	string get_functionId ()
java	String get_functionId ()
py	get_functionId ()
php	function get_functionId ()
ts	async get_functionId (): Promise<string>
es	async get_functionId ()
dnp	string get_functionId ()
cp	string get_functionId ()

For example `relay1`

Returns :

a string that identifies the current sensor (ex: `relay1`)

On failure, throws an exception or returns `YCurrent.FUNCTIONID_INVALID`.

current→**get_hardwareId()****YCurrent****current**→**hardwareId()**

Returns the unique hardware identifier of the current sensor in the form `SERIAL . FUNCTIONID`.

js	function get_hardwareId ()
cpp	string get_hardwareId ()
m	-(NSString*) hardwareId
vb	function get_hardwareId () As String
cs	string get_hardwareId ()
java	String get_hardwareId ()
py	get_hardwareId ()
php	function get_hardwareId ()
ts	async get_hardwareId (): Promise<string>
es	async get_hardwareId ()
dnp	string get_hardwareId ()
cp	string get_hardwareId ()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the current sensor (for example `RELAYL01-123456 . relay1`).

Returns :

a string that uniquely identifies the current sensor (ex: `RELAYL01-123456 . relay1`)

On failure, throws an exception or returns `YCurrent . HARDWAREID_INVALID`.

current→**get_highestValue()****YCurrent****current**→**highestValue()**

Returns the maximal value observed for the current since the device was started.

js	function get_highestValue ()
cpp	double get_highestValue ()
m	-(double) highestValue
pas	double get_highestValue (): double
vb	function get_highestValue () As Double
cs	double get_highestValue ()
java	double get_highestValue ()
uwp	async Task<double> get_highestValue ()
py	get_highestValue ()
php	function get_highestValue ()
ts	async get_highestValue (): Promise<number>
es	async get_highestValue ()
dnp	double get_highestValue ()
cp	double get_highestValue ()
cmd	YCurrent target get_highestValue

Can be reset to an arbitrary value thanks to `set_highestValue()`.

Returns :

a floating point number corresponding to the maximal value observed for the current since the device was started

On failure, throws an exception or returns `YCurrent.HIGHESTVALUE_INVALID`.

current→get_logFrequency()**YCurrent****current→logFrequency()**

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

js	function get_logFrequency ()
cpp	string get_logFrequency ()
m	-(NSString*) logFrequency
pas	string get_logFrequency (): string
vb	function get_logFrequency () As String
cs	string get_logFrequency ()
java	String get_logFrequency ()
uwp	async Task<string> get_logFrequency ()
py	get_logFrequency ()
php	function get_logFrequency ()
ts	async get_logFrequency (): Promise<string>
es	async get_logFrequency ()
dnp	string get_logFrequency ()
cp	string get_logFrequency ()
cmd	YCurrent target get_logFrequency

Returns :

a string corresponding to the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory

On failure, throws an exception or returns YCurrent.LOGFREQUENCY_INVALID.

current→get_logicalName()**YCurrent****current→logicalName()**

Returns the logical name of the current sensor.

js	function get_logicalName ()
cpp	string get_logicalName ()
m	-(NSString*) logicalName
pas	string get_logicalName (): string
vb	function get_logicalName () As String
cs	string get_logicalName ()
java	String get_logicalName ()
uwp	async Task<string> get_logicalName ()
py	get_logicalName ()
php	function get_logicalName ()
ts	async get_logicalName (): Promise<string>
es	async get_logicalName ()
dnp	string get_logicalName ()
cp	string get_logicalName ()
cmd	YCurrent target get_logicalName

Returns :

a string corresponding to the logical name of the current sensor.

On failure, throws an exception or returns YCurrent.LOGICALNAME_INVALID.

current→**get_lowestValue()****YCurrent****current**→**lowestValue()**

Returns the minimal value observed for the current since the device was started.

js	function get_lowestValue ()
cpp	double get_lowestValue ()
m	-(double) lowestValue
pas	double get_lowestValue (): double
vb	function get_lowestValue () As Double
cs	double get_lowestValue ()
java	double get_lowestValue ()
uwp	async Task<double> get_lowestValue ()
py	get_lowestValue ()
php	function get_lowestValue ()
ts	async get_lowestValue (): Promise<number>
es	async get_lowestValue ()
dnp	double get_lowestValue ()
cp	double get_lowestValue ()
cmd	YCurrent target get_lowestValue

Can be reset to an arbitrary value thanks to **set_lowestValue()**.

Returns :

a floating point number corresponding to the minimal value observed for the current since the device was started

On failure, throws an exception or returns **YCurrent . LOWESTVALUE_INVALID**.

current→get_module()**YCurrent****current→module()**

Gets the YModule object for the device on which the function is located.

js	function get_module ()
cpp	YModule * get_module ()
m	-(YModule*) module
pas	TYModule get_module (): TYModule
vb	function get_module () As YModule
cs	YModule get_module ()
java	YModule get_module ()
py	get_module ()
php	function get_module ()
ts	async get_module (): Promise<YModule>
es	async get_module ()
dnp	YModuleProxy get_module ()
cp	YModuleProxy * get_module ()

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

current→**get_module_async()****YCurrent****current**→**module_async()**

Gets the YModule object for the device on which the function is located (asynchronous version).

```
js function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as on-line.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking Firefox JavaScript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous JavaScript calls for more details.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

current→get_recordedData()**YCurrent****current→recordedData()**

Retrieves a `YDataSet` object holding historical data for this sensor, for a specified time interval.

js	<code>function get_recordedData(startTime, endTime)</code>
cpp	<code>YDataSet get_recordedData(double startTime, double endTime)</code>
m	<code>-(YDataSet*) recordedData : (double) startTime : (double) endTime</code>
pas	<code>TYDataSet get_recordedData(startTime: double, endTime: double): TYDataSet</code>
vb	<code>function get_recordedData(ByVal startTime As Double, ByVal endTime As Double) As YDataSet</code>
cs	<code>YDataSet get_recordedData(double startTime, double endTime)</code>
java	<code>YDataSet get_recordedData(double startTime, double endTime)</code>
uwp	<code>async Task<YDataSet> get_recordedData(double startTime, double endTime)</code>
py	<code>get_recordedData(startTime, endTime)</code>
php	<code>function get_recordedData(\$startTime, \$endTime)</code>
ts	<code>async get_recordedData(startTime: number, endTime: number): Promise<YDataSet></code>
es	<code>async get_recordedData(startTime, endTime)</code>
dnp	<code>YDataSetProxy get_recordedData(double startTime, double endTime)</code>
cp	<code>YDataSetProxy* get_recordedData(double startTime, double endTime)</code>
cmd	<code>YCurrent target get_recordedData startTime endTime</code>

The measures will be retrieved from the data logger, which must have been turned on at the desired time. See the documentation of the `YDataSet` class for information on how to get an overview of the recorded data, and how to load progressively a large set of measures from the data logger.

This function only works if the device uses a recent firmware, as `YDataSet` objects are not supported by firmwares older than version 13000.

Parameters :

- startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without initial limit.
- endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without ending limit.

Returns :

an instance of `YDataSet`, providing access to historical data. Past measures can be loaded progressively using methods from the `YDataSet` object.

current→get_reportFrequency()**YCurrent****current→reportFrequency()**

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

js	function get_reportFrequency ()
cpp	string get_reportFrequency ()
m	-(NSString*) reportFrequency
pas	string get_reportFrequency (): string
vb	function get_reportFrequency () As String
cs	string get_reportFrequency ()
java	String get_reportFrequency ()
uwp	async Task<string> get_reportFrequency ()
py	get_reportFrequency ()
php	function get_reportFrequency ()
ts	async get_reportFrequency (): Promise<string>
es	async get_reportFrequency ()
dnp	string get_reportFrequency ()
cp	string get_reportFrequency ()
cmd	YCurrent target get_reportFrequency

Returns :

a string corresponding to the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function

On failure, throws an exception or returns YCurrent.REPORTFREQUENCY_INVALID.

current→**get_resolution()****YCurrent****current**→**resolution()**

Returns the resolution of the measured values.

js	function get_resolution ()
cpp	double get_resolution ()
m	-(double) resolution
pas	double get_resolution (): double
vb	function get_resolution () As Double
cs	double get_resolution ()
java	double get_resolution ()
uwp	async Task<double> get_resolution ()
py	get_resolution ()
php	function get_resolution ()
ts	async get_resolution (): Promise<number>
es	async get_resolution ()
dnp	double get_resolution ()
cp	double get_resolution ()
cmd	YCurrent target get_resolution

The resolution corresponds to the numerical precision of the measures, which is not always the same as the actual precision of the sensor. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

a floating point number corresponding to the resolution of the measured values

On failure, throws an exception or returns `YCurrent.RESOLUTION_INVALID`.

current→get_sensorState()**YCurrent****current→sensorState()**

Returns the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

js	function get_sensorState ()
cpp	int get_sensorState ()
m	-(int) sensorState
pas	LongInt get_sensorState (): LongInt
vb	function get_sensorState () As Integer
cs	int get_sensorState ()
java	int get_sensorState ()
uwp	async Task<int> get_sensorState ()
py	get_sensorState ()
php	function get_sensorState ()
ts	async get_sensorState (): Promise<number>
es	async get_sensorState ()
dnp	int get_sensorState ()
cp	int get_sensorState ()
cmd	YCurrent target get_sensorState

Returns :

an integer corresponding to the sensor health state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now

On failure, throws an exception or returns `YCurrent.SENSORSTATE_INVALID`.

current→get_serialNumber()**YCurrent****current→serialNumber()**

Returns the serial number of the module, as set by the factory.

js	function get_serialNumber ()
cpp	string get_serialNumber ()
m	-(NSString*) serialNumber
pas	string get_serialNumber (): string
vb	function get_serialNumber () As String
cs	string get_serialNumber ()
java	String get_serialNumber ()
uwp	async Task<string> get_serialNumber ()
py	get_serialNumber ()
php	function get_serialNumber ()
ts	async get_serialNumber (): Promise<string>
es	async get_serialNumber ()
dnp	string get_serialNumber ()
cp	string get_serialNumber ()
cmd	YCurrent target get_serialNumber

Returns :

a string corresponding to the serial number of the module, as set by the factory.

On failure, throws an exception or returns YFunction.SERIALNUMBER_INVALID.

current→get_unit()**YCurrent****current→unit()**

Returns the measuring unit for the current.

js	function get_unit ()
cpp	string get_unit ()
m	-(NSString*) unit
pas	string get_unit (): string
vb	function get_unit () As String
cs	string get_unit ()
java	String get_unit ()
uwp	async Task<string> get_unit ()
py	get_unit ()
php	function get_unit ()
ts	async get_unit (): Promise<string>
es	async get_unit ()
dnp	string get_unit ()
cp	string get_unit ()
cmd	YCurrent target get_unit

Returns :

a string corresponding to the measuring unit for the current

On failure, throws an exception or returns YCurrent . UNIT_INVALID.

current→**get_userData()****YCurrent****current**→**userData()**

Returns the value of the userData attribute, as previously stored using method `set_userData`.

js	<code>function get_userData()</code>
cpp	<code>void * get_userData()</code>
m	<code>-(id) userData</code>
pas	<code>Tobject get_userData(): Tobject</code>
vb	<code>function get_userData() As Object</code>
cs	<code>object get_userData()</code>
java	<code>Object get_userData()</code>
py	<code>get_userData()</code>
php	<code>function get_userData()</code>
ts	<code>async get_userData(): Promise<object null></code>
es	<code>async get_userData()</code>

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

current→isOnline()**YCurrent**

Checks if the current sensor is currently reachable, without raising any error.

js	<code>function isOnline()</code>
cpp	<code>bool isOnline()</code>
m	<code>-(BOOL) isOnline</code>
pas	<code>boolean isOnline(): boolean</code>
vb	<code>function isOnline() As Boolean</code>
cs	<code>bool isOnline()</code>
java	<code>boolean isOnline()</code>
py	<code>isOnline()</code>
php	<code>function isOnline()</code>
ts	<code>async isOnline(): Promise<boolean></code>
es	<code>async isOnline()</code>
dnp	<code>bool isOnline()</code>
cp	<code>bool isOnline()</code>

If there is a cached value for the current sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the current sensor.

Returns :

`true` if the current sensor can be reached, and `false` otherwise

current→isOnline_async()**YCurrent**

Checks if the current sensor is currently reachable, without raising any error (asynchronous version).

```
js function isOnline_async( callback, context)
```

If there is a cached value for the current sensor in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

current→isReadOnly()**YCurrent**

Test if the function is readOnly.

cpp	<code>bool isReadOnly()</code>
m	<code>-(bool) isReadOnly</code>
pas	<code>boolean isReadOnly(): boolean</code>
vb	<code>function isReadOnly() As Boolean</code>
cs	<code>bool isReadOnly()</code>
java	<code>boolean isReadOnly()</code>
uwp	<code>async Task<bool> isReadOnly()</code>
py	<code>isReadOnly()</code>
php	<code>function isReadOnly()</code>
ts	<code>async isReadOnly(): Promise<boolean></code>
es	<code>async isReadOnly()</code>
dnp	<code>bool isReadOnly()</code>
cp	<code>bool isReadOnly()</code>
cmd	<code>YCurrent target isReadOnly</code>

Return `true` if the function is write protected or that the function is not available.

Returns :

`true` if the function is readOnly or not online.

current→isSensorReady()**YCurrent**

Checks if the sensor is currently able to provide an up-to-date measure.

cmd **YCurrent target isSensorReady**

Returns false if the device is unreachable, or if the sensor does not have a current measure to transmit. No exception is raised if there is an error while trying to contact the device hosting \$THEFUNCTION\$.

Returns :

`true` if the sensor can provide an up-to-date measure, and `false` otherwise

current→load()**YCurrent**

Preloads the current sensor cache with a specified validity duration.

js	function load (msValidity)
cpp	YRETCODE load (int msValidity)
m	-(YRETCODE) load : (u64) msValidity
pas	YRETCODE load (msValidity : u64): YRETCODE
vb	function load (ByVal msValidity As Long) As YRETCODE
cs	YRETCODE load (ulong msValidity)
java	int load (long msValidity)
py	load (msValidity)
php	function load (\$msValidity)
ts	async load (msValidity : number): Promise<number>
es	async load (msValidity)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI . SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

current→loadAttribute()**YCurrent**

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

js	function loadAttribute (attrName)
cpp	string loadAttribute (string attrName)
m	-(NSString*) loadAttribute : (NSString*) attrName
pas	string loadAttribute (attrName : string): string
vb	function loadAttribute (ByVal attrName As String) As String
cs	string loadAttribute (string attrName)
java	String loadAttribute (String attrName)
uwp	async Task<string> loadAttribute (string attrName)
py	loadAttribute (attrName)
php	function loadAttribute (\$ attrName)
ts	async loadAttribute (attrName : string): Promise<string>
es	async loadAttribute (attrName)
dnp	string loadAttribute (string attrName)
cp	string loadAttribute (string attrName)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

current→loadCalibrationPoints()**YCurrent**

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

```

js  function loadCalibrationPoints( rawValues, refValues)
cpp int loadCalibrationPoints( vector<double> rawValues,
                               vector<double> refValues)
m   -(int) loadCalibrationPoints : (NSMutableArray*) rawValues
                               : (NSMutableArray*) refValues
pas LongInt loadCalibrationPoints( var rawValues: TDoubleArray,
                                   var refValues: TDoubleArray): LongInt
vb  procedure loadCalibrationPoints( ByVal rawValues As List(Of)
cs  int loadCalibrationPoints( List<double> rawValues,
                               List<double> refValues)
java int loadCalibrationPoints( ArrayList<Double> rawValues,
                               ArrayList<Double> refValues)
uwp async Task<int> loadCalibrationPoints( List<double> rawValues,
                                           List<double> refValues)
py  loadCalibrationPoints( rawValues, refValues)
php function loadCalibrationPoints( &$rawValues, &$refValues)
ts  async loadCalibrationPoints( rawValues: number[], refValues: number[]): Promise<number>
es  async loadCalibrationPoints( rawValues, refValues)
cmd YCurrent target loadCalibrationPoints rawValues refValues

```

Parameters :

rawValues array of floating point numbers, that will be filled by the function with the raw sensor values for the correction points.

refValues array of floating point numbers, that will be filled by the function with the desired values for the correction points.

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

current→load_async()**YCurrent**

Preloads the current sensor cache with a specified validity duration (asynchronous version).

```
js function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking the JavaScript virtual machine.

Parameters :

- msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds
- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or `YAPI . SUCCESS`)
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

current→muteValueCallbacks()**YCurrent**

Disables the propagation of every new advertised value to the parent hub.

js	function muteValueCallbacks ()
cpp	int muteValueCallbacks ()
m	-(int) muteValueCallbacks
pas	LongInt muteValueCallbacks (): LongInt
vb	function muteValueCallbacks () As Integer
cs	int muteValueCallbacks ()
java	int muteValueCallbacks ()
uwp	async Task<int> muteValueCallbacks ()
py	muteValueCallbacks ()
php	function muteValueCallbacks ()
ts	async muteValueCallbacks (): Promise<number>
es	async muteValueCallbacks ()
dnp	int muteValueCallbacks ()
cp	int muteValueCallbacks ()
cmd	YCurrent target muteValueCallbacks

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

YAPI . SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

current→nextCurrent()**YCurrent**

Continues the enumeration of current sensors started using `yFirstCurrent()`.

js	<code>function nextCurrent()</code>
cpp	<code>YCurrent * nextCurrent()</code>
m	<code>-(nullable YCurrent*) nextCurrent</code>
pas	<code>TYCurrent nextCurrent(): TYCurrent</code>
vb	<code>function nextCurrent() As YCurrent</code>
cs	<code>YCurrent nextCurrent()</code>
java	<code>YCurrent nextCurrent()</code>
uwp	<code>YCurrent nextCurrent()</code>
py	<code>nextCurrent()</code>
php	<code>function nextCurrent()</code>
ts	<code>nextCurrent(): YCurrent null</code>
es	<code>nextCurrent()</code>

Caution: You can't make any assumption about the returned current sensors order. If you want to find a specific a current sensor, use `Current.findCurrent()` and a `hardwareID` or a logical name.

Returns :

a pointer to a `YCurrent` object, corresponding to a current sensor currently online, or a `null` pointer if there are no more current sensors to enumerate.

current→registerTimedReportCallback()**YCurrent**

Registers the callback function that is invoked on every periodic timed notification.

js	function registerTimedReportCallback (callback)
cpp	int registerTimedReportCallback (YCurrentTimedReportCallback callback)
m	-(int) registerTimedReportCallback : (YCurrentTimedReportCallback _Nullable) callback
pas	LongInt registerTimedReportCallback (callback : TYCurrentTimedReportCallback): LongInt
vb	function registerTimedReportCallback (ByVal callback As YCurrentTimedReportCallback) As Integer
cs	int registerTimedReportCallback (TimedReportCallback callback)
java	int registerTimedReportCallback (TimedReportCallback callback)
uwp	async Task<int> registerTimedReportCallback (TimedReportCallback callback)
py	registerTimedReportCallback (callback)
php	function registerTimedReportCallback (\$ callback)
ts	async registerTimedReportCallback (callback : YCurrentTimedReportCallback null): Promise<number>
es	async registerTimedReportCallback (callback)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and an `YMeasure` object describing the new advertised value.

current→registerValueCallback()**YCurrent**

Registers the callback function that is invoked on every change of advertised value.

js	function registerValueCallback (callback)
cpp	int registerValueCallback (YCurrentValueCallback callback)
m	-(int) registerValueCallback : (YCurrentValueCallback _Nullable) callback
pas	LongInt registerValueCallback (callback : TYCurrentValueCallback): LongInt
vb	function registerValueCallback (ByVal callback As YCurrentValueCallback) As Integer
cs	int registerValueCallback (ValueCallback callback)
java	int registerValueCallback (UpdateCallback callback)
uwp	async Task<int> registerValueCallback (ValueCallback callback)
py	registerValueCallback (callback)
php	function registerValueCallback (\$callback)
ts	async registerValueCallback (callback : YCurrentValueCallback null): Promise<number>
es	async registerValueCallback (callback)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

current→**set_advMode()****YCurrent****current**→**setAdvMode()**

Changes the measuring mode used for the advertised value pushed to the parent hub.

js	function set_advMode (newval)
cpp	int set_advMode (Y_ADVMODE_enum newval)
m	-(int) setAdvMode : (Y_ADVMODE_enum) newval
pas	integer set_advMode (newval : Integer): integer
vb	function set_advMode (ByVal newval As Integer) As Integer
cs	int set_advMode (int newval)
java	int set_advMode (int newval)
uwp	async Task<int> set_advMode (int newval)
py	set_advMode (newval)
php	function set_advMode (\$ newval)
ts	async set_advMode (newval : YSensor_AdvMode): Promise<number>
es	async set_advMode (newval)
dnp	int set_advMode (int newval)
cp	int set_advMode (int newval)
cmd	YCurrent target set_advMode newval

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a value among `YCurrent.ADVMODE_IMMEDIATE`, `YCurrent.ADVMODE_PERIOD_AVG`, `YCurrent.ADVMODE_PERIOD_MIN` and `YCurrent.ADVMODE_PERIOD_MAX` corresponding to the measuring mode used for the advertised value pushed to the parent hub

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

current→set_enabled()**YCurrent****current→setEnabled()**

Changes the activation state of this voltage input.

js	function set_enabled (newval)
cpp	int set_enabled (Y_ENABLED_enum newval)
m	-(int) setEnabled : (Y_ENABLED_enum) newval
pas	integer set_enabled (newval : Integer): integer
vb	function set_enabled (ByVal newval As Integer) As Integer
cs	int set_enabled (int newval)
java	int set_enabled (int newval)
uwp	async Task<int> set_enabled (int newval)
py	set_enabled (newval)
php	function set_enabled (\$ newval)
ts	async set_enabled (newval : YCurrent_Enabled): Promise<number>
es	async set_enabled (newval)
dnp	int set_enabled (int newval)
cp	int set_enabled (int newval)
cmd	YCurrent target set_enabled newval

When AC measurements are disabled, the device will always assume a DC signal, and vice-versa. When both AC and DC measurements are active, the device switches between AC and DC mode based on the relative amplitude of variations compared to the average value. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval either `YCurrent.ENABLED_FALSE` or `YCurrent.ENABLED_TRUE`, according to the activation state of this voltage input

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

current→set_highestValue()**YCurrent****current→setHighestValue()**

Changes the recorded maximal value observed.

js	function set_highestValue (newval)
cpp	int set_highestValue (double newval)
m	-(int) setHighestValue : (double) newval
pas	integer set_highestValue (newval : double): integer
vb	function set_highestValue (ByVal newval As Double) As Integer
cs	int set_highestValue (double newval)
java	int set_highestValue (double newval)
uwp	async Task<int> set_highestValue (double newval)
py	set_highestValue (newval)
php	function set_highestValue (\$newval)
ts	async set_highestValue (newval : number): Promise<number>
es	async set_highestValue (newval)
dnp	int set_highestValue (double newval)
cp	int set_highestValue (double newval)
cmd	YCurrent target set_highestValue newval

Can be used to reset the value returned by get_lowestValue().

Parameters :

newval a floating point number corresponding to the recorded maximal value observed

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

current→set_logFrequency()**current→setLogFrequency()**

Changes the datalogger recording frequency for this function.

js	function set_logFrequency (newval)
cpp	int set_logFrequency (string newval)
m	-(int) setLogFrequency : (NSString*) newval
pas	integer set_logFrequency (newval : string): integer
vb	function set_logFrequency (ByVal newval As String) As Integer
cs	int set_logFrequency (string newval)
java	int set_logFrequency (String newval)
uwp	async Task<int> set_logFrequency (string newval)
py	set_logFrequency (newval)
php	function set_logFrequency (\$ newval)
ts	async set_logFrequency (newval : string): Promise<number>
es	async set_logFrequency (newval)
dnp	int set_logFrequency (string newval)
cp	int set_logFrequency (string newval)
cmd	YCurrent target set_logFrequency newval

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (eg. "4/h"). To disable recording for this function, use the value "OFF". Note that setting the datalogger recording frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the datalogger recording frequency for this function

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

current→set_logicalName()**YCurrent****current→setLogicalName()**

Changes the logical name of the current sensor.

js	function set_logicalName (newval)
cpp	int set_logicalName (string newval)
m	-(int) setLogicalName : (NSString*) newval
pas	integer set_logicalName (newval : string): integer
vb	function set_logicalName (ByVal newval As String) As Integer
cs	int set_logicalName (string newval)
java	int set_logicalName (String newval)
uwp	async Task<int> set_logicalName (string newval)
py	set_logicalName (newval)
php	function set_logicalName (\$ newval)
ts	async set_logicalName (newval : string): Promise<number>
es	async set_logicalName (newval)
dnp	int set_logicalName (string newval)
cp	int set_logicalName (string newval)
cmd	YCurrent target set_logicalName newval

You can use `yCheckLogicalName( )` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the current sensor.

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

current→**set_lowestValue()****YCurrent****current**→**setLowestValue()**

Changes the recorded minimal value observed.

js	function set_lowestValue (newval)
cpp	int set_lowestValue (double newval)
m	-(int) setLowestValue : (double) newval
pas	integer set_lowestValue (newval : double): integer
vb	function set_lowestValue (ByVal newval As Double) As Integer
cs	int set_lowestValue (double newval)
java	int set_lowestValue (double newval)
uwp	async Task<int> set_lowestValue (double newval)
py	set_lowestValue (newval)
php	function set_lowestValue (\$newval)
ts	async set_lowestValue (newval : number): Promise<number>
es	async set_lowestValue (newval)
dnp	int set_lowestValue (double newval)
cp	int set_lowestValue (double newval)
cmd	YCurrent target set_lowestValue newval

Can be used to reset the value returned by `get_lowestValue()`.

Parameters :

newval a floating point number corresponding to the recorded minimal value observed

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

current→set_reportFrequency()**YCurrent****current→setReportFrequency()**

Changes the timed value notification frequency for this function.

js	function set_reportFrequency (newval)
cpp	int set_reportFrequency (string newval)
m	-(int) setReportFrequency : (NSString*) newval
pas	integer set_reportFrequency (newval : string): integer
vb	function set_reportFrequency (ByVal newval As String) As Integer
cs	int set_reportFrequency (string newval)
java	int set_reportFrequency (String newval)
uwp	async Task<int> set_reportFrequency (string newval)
py	set_reportFrequency (newval)
php	function set_reportFrequency (\$ newval)
ts	async set_reportFrequency (newval : string): Promise<number>
es	async set_reportFrequency (newval)
dnp	int set_reportFrequency (string newval)
cp	int set_reportFrequency (string newval)
cmd	YCurrent target set_reportFrequency newval

The frequency can be specified as samples per second, as sample per minute (for instance "15/m") or in samples per hour (e.g. "4/h"). To disable timed value notifications for this function, use the value "OFF". Note that setting the timed value notification frequency to a greater value than the sensor native sampling frequency is useless, and even counterproductive: those two frequencies are not related. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the timed value notification frequency for this function

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

current→**set_resolution()****current**→**setResolution()**

Changes the resolution of the measured physical values.

js	function set_resolution (newval)
cpp	int set_resolution (double newval)
m	-(int) setResolution : (double) newval
pas	integer set_resolution (newval : double): integer
vb	function set_resolution (ByVal newval As Double) As Integer
cs	int set_resolution (double newval)
java	int set_resolution (double newval)
uwp	async Task<int> set_resolution (double newval)
py	set_resolution (newval)
php	function set_resolution (\$newval)
ts	async set_resolution (newval : number): Promise<number>
es	async set_resolution (newval)
dnp	int set_resolution (double newval)
cp	int set_resolution (double newval)
cmd	YCurrent target set_resolution newval

The resolution corresponds to the numerical precision when displaying value. It does not change the precision of the measure itself. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a floating point number corresponding to the resolution of the measured physical values

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

current→set_userData()**YCurrent****current→setUserData()**

Stores a user context provided as argument in the userData attribute of the function.

js	function set_userData (data)
cpp	void set_userData (void * data)
m	-(void) setUserData : (id) data
pas	set_userData (data : Tobject)
vb	procedure set_userData (ByVal data As Object)
cs	void set_userData (object data)
java	void set_userData (Object data)
py	set_userData (data)
php	function set_userData (\$data)
ts	async set_userData (data : object null): Promise<void>
es	async set_userData (data)

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

current→startDataLogger()**YCurrent**

Starts the data logger on the device.

js	function startDataLogger ()
cpp	int startDataLogger ()
m	-(int) startDataLogger
pas	LongInt startDataLogger (): LongInt
vb	function startDataLogger () As Integer
cs	int startDataLogger ()
java	int startDataLogger ()
uwp	async Task<int> startDataLogger ()
py	startDataLogger ()
php	function startDataLogger ()
ts	async startDataLogger (): Promise<number>
es	async startDataLogger ()
dnp	int startDataLogger ()
cp	int startDataLogger ()
cmd	YCurrent target startDataLogger

Note that the data logger will only save the measures on this sensor if the logFrequency is not set to "OFF".

Returns :

YAPI . SUCCESS if the call succeeds.

current→stopDataLogger()**YCurrent**

Stops the datalogger on the device.

js	function stopDataLogger ()
cpp	int stopDataLogger ()
m	-(int) stopDataLogger
pas	LongInt stopDataLogger (): LongInt
vb	function stopDataLogger () As Integer
cs	int stopDataLogger ()
java	int stopDataLogger ()
uwp	async Task<int> stopDataLogger ()
py	stopDataLogger ()
php	function stopDataLogger ()
ts	async stopDataLogger (): Promise<number>
es	async stopDataLogger ()
dnp	int stopDataLogger ()
cp	int stopDataLogger ()
cmd	YCurrent target stopDataLogger

Returns :

YAPI . SUCCESS if the call succeeds.

current→unmuteValueCallbacks()**YCurrent**

Re-enables the propagation of every new advertised value to the parent hub.

js	function unmuteValueCallbacks ()
cpp	int unmuteValueCallbacks ()
m	-(int) unmuteValueCallbacks
pas	LongInt unmuteValueCallbacks (): LongInt
vb	function unmuteValueCallbacks () As Integer
cs	int unmuteValueCallbacks ()
java	int unmuteValueCallbacks ()
uwp	async Task<int> unmuteValueCallbacks ()
py	unmuteValueCallbacks ()
php	function unmuteValueCallbacks ()
ts	async unmuteValueCallbacks (): Promise<number>
es	async unmuteValueCallbacks ()
dnp	int unmuteValueCallbacks ()
cp	int unmuteValueCallbacks ()
cmd	YCurrent target unmuteValueCallbacks

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

YAPI . SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

current→wait_async()**YCurrent**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

js	<code>function wait_async(callback, context)</code>

ts	<code>wait_async(callback: Function, context: object)</code>

es	<code>wait_async(callback, context)</code>

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

24.6. 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:

js	<code><script type='text/javascript' src='yocto_module.js'></script></code>
cpp	<code>#include "yocto_module.h"</code>
m	<code>#import "yocto_module.h"</code>
pas	<code>uses yocto_module;</code>
vb	<code>yocto_module.vb</code>
cs	<code>yocto_module.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YDataLogger;</code>
uwp	<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>
es	<code>in HTML: <script src='../lib/yocto_module.js'></script> in node.js: require('yoctolib-es2017/yocto_module.js');</code>
dnf	<code>import YoctoProxyAPI.YDataLoggerProxy</code>
cp	<code>#include "yocto_module_proxy.h"</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.

YDataLogger.FindDataLoggerInContext(yctx, func)

Retrieves a data logger for a given identifier in a YAPI context.

YDataLogger.FirstDataLogger()

Starts the enumeration of data loggers currently accessible.

YDataLogger.FirstDataLoggerInContext(yctx)

Starts the enumeration of data loggers currently accessible.

YDataLogger.GetSimilarFunctions()

Enumerates all functions of type DataLogger available on the devices currently reachable by the library, and returns their unique hardware ID.

YDataLogger properties

datalogger→AdvertisedValue *[read-only]*

Short string representing the current state of the function.

datalogger→AutoStart *[writable]*

Default activation state of the data logger on power up.

datalogger→BeaconDriven *[writable]*

True if the data logger is synchronised with the localization beacon.

datalogger→FriendlyName *[read-only]*

Global identifier of the function in the format `MODULE_NAME . FUNCTION_NAME`.

datalogger→FunctionId *[read-only]*

Hardware identifier of the data logger, without reference to the module.

datalogger→HardwareId *[read-only]*

Unique hardware identifier of the function in the form `SERIAL . FUNCTIONID`.

datalogger→IsOnline *[read-only]*

Checks if the function is currently reachable.

datalogger→LogicalName *[writable]*

Logical name of the function.

datalogger→Recording *[writable]*

Current activation state of the data logger.

datalogger→SerialNumber *[read-only]*

Serial number of the module, as set by the factory.

YDataLogger methods

datalogger→clearCache()

Invalidates the cache.

datalogger→describe()

Returns a short text that describes unambiguously the instance of the data logger in the form `TYPE (NAME) = SERIAL . FUNCTIONID`.

datalogger→forgetAllDataStreams()

Clears the data logger memory and discards all recorded data streams.

datalogger→get_advertisedValue()

Returns the current value of the data logger (no more than 6 characters).

datalogger→get_autoStart()

Returns the default activation state of the data logger on power up.

datalogger→get_beaconDriven()

Returns true if the data logger is synchronised with the localization beacon.

datalogger→get_currentRunIndex()

Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.

datalogger→get_dataSets()

Returns a list of `YDataSet` objects that can be used to retrieve all measures stored by the data logger.

datalogger→get_dataStreams(v)

Builds a list of all data streams hold by the data logger (legacy method).

datalogger→get_errorMessage()

Returns the error message of the latest error with the data logger.

datalogger→get_errorType()

Returns the numerical error code of the latest error with the data logger.

datalogger→get_friendlyName()

Returns a global identifier of the data logger in the format `MODULE_NAME . FUNCTION_NAME`.

datalogger→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

datalogger→get_functionId()

Returns the hardware identifier of the data logger, without reference to the module.

datalogger→get_hardwareId()

Returns the unique hardware identifier of the data logger in the form `SERIAL . FUNCTIONID`.

`datalogger→get_logicalName()`

Returns the logical name of the data logger.

`datalogger→get_module()`

Gets the `YModule` object for the device on which the function is located.

`datalogger→get_module_async(callback, context)`

Gets the `YModule` object for the device on which the function is located (asynchronous version).

`datalogger→get_recording()`

Returns the current activation state of the data logger.

`datalogger→get_serialNumber()`

Returns the serial number of the module, as set by the factory.

`datalogger→get_timeUTC()`

Returns the Unix timestamp for current UTC time, if known.

`datalogger→get_usage()`

Returns the percentage of datalogger memory in use.

`datalogger→get_userData()`

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

`datalogger→isOnline()`

Checks if the data logger is currently reachable, without raising any error.

`datalogger→isOnline_async(callback, context)`

Checks if the data logger is currently reachable, without raising any error (asynchronous version).

`datalogger→isReadOnly()`

Test if the function is `readOnly`.

`datalogger→load(msValidity)`

Preloads the data logger cache with a specified validity duration.

`datalogger→loadAttribute(attrName)`

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

`datalogger→load_async(msValidity, callback, context)`

Preloads the data logger cache with a specified validity duration (asynchronous version).

`datalogger→muteValueCallbacks()`

Disables the propagation of every new advertised value to the parent hub.

`datalogger→nextDataLogger()`

Continues the enumeration of data loggers started using `yFirstDataLogger()`.

`datalogger→registerValueCallback(callback)`

Registers the callback function that is invoked on every change of advertised value.

`datalogger→set_autoStart(newval)`

Changes the default activation state of the data logger on power up.

`datalogger→set_beaconDriven(newval)`

Changes the type of synchronisation of the data logger.

`datalogger→set_logicalName(newval)`

Changes the logical name of the data logger.

`datalogger→set_recording(newval)`

Changes the activation state of the data logger to start/stop recording data.

`datalogger→set_timeUTC(newval)`

Changes the current UTC time reference used for recorded data.

`datalogger→set_userdata(data)`

Stores a user context provided as argument in the `userData` attribute of the function.

`datalogger→unmuteValueCallbacks()`

Re-enables the propagation of every new advertised value to the parent hub.

`datalogger→wait_async(callback, context)`

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YDataLogger.FindDataLogger()

YDataLogger.FindDataLogger()

YDataLogger

Retrieves a data logger for a given identifier.

js	function yFindDataLogger (func)
cpp	YDataLogger* FindDataLogger (string func)
m	+(YDataLogger*) FindDataLogger : (NSString*) func
pas	TYDataLogger yFindDataLogger (func : string): TYDataLogger
vb	function FindDataLogger (ByVal func As String) As YDataLogger
cs	static YDataLogger FindDataLogger (string func)
java	static YDataLogger FindDataLogger (String func)
uwp	static YDataLogger FindDataLogger (string func)
py	FindDataLogger (func)
php	function FindDataLogger (\$func)
ts	static FindDataLogger (func : string): YDataLogger
es	static FindDataLogger (func)
dnp	static YDataLoggerProxy FindDataLogger (string func)
cp	static YDataLoggerProxy * FindDataLogger (string func)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the data logger is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDataLogger.isOnline()` to test if the data logger is indeed online at a given time. In case of ambiguity when looking for a data logger by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

If a call to this object's `is_online()` method returns `FALSE` although you are certain that the matching device is plugged, make sure that you did call `registerHub()` at application initialization time.

Parameters :

func a string that uniquely characterizes the data logger, for instance `RX420MA1.dataLogger`.

Returns :

a `YDataLogger` object allowing you to drive the data logger.

YDataLogger.FindDataLoggerInContext()**YDataLogger****YDataLogger.FindDataLoggerInContext()**

Retrieves a data logger for a given identifier in a YAPI context.

```

java static YDataLogger FindDataLoggerInContext( YAPIContext yctx,
                                                String func)
uwp static YDataLogger FindDataLoggerInContext( YAPIContext yctx,
                                                string func)
ts static FindDataLoggerInContext( yctx: YAPIContext, func: string): YDataLogger
es static FindDataLoggerInContext( yctx, func)

```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the data logger is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDataLogger.isOnline()` to test if the data logger is indeed online at a given time. In case of ambiguity when looking for a data logger by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the data logger, for instance `RX420MA1.dataLogger`.

Returns :

a `YDataLogger` object allowing you to drive the data logger.

YDataLogger.FirstDataLogger()**YDataLogger****YDataLogger.FirstDataLogger()**

Starts the enumeration of data loggers currently accessible.

js	function yFirstDataLogger ()
cpp	YDataLogger * FirstDataLogger ()
m	+(YDataLogger*) FirstDataLogger
pas	TYDataLogger yFirstDataLogger (): TYDataLogger
vb	function FirstDataLogger () As YDataLogger
cs	static YDataLogger FirstDataLogger ()
java	static YDataLogger FirstDataLogger ()
uwp	static YDataLogger FirstDataLogger ()
py	FirstDataLogger ()
php	function FirstDataLogger ()
ts	static FirstDataLogger (): YDataLogger null
es	static FirstDataLogger ()

Use the method `YDataLogger.nextDataLogger( )` to iterate on next data loggers.

Returns :

a pointer to a `YDataLogger` object, corresponding to the first data logger currently online, or a `null` pointer if there are none.

YDataLogger.FirstDataLoggerInContext()**YDataLogger****YDataLogger.FirstDataLoggerInContext()**

Starts the enumeration of data loggers currently accessible.

java	static YDataLogger FirstDataLoggerInContext (YAPIContext yctx)
uwp	static YDataLogger FirstDataLoggerInContext (YAPIContext yctx)
ts	static FirstDataLoggerInContext (yctx : YAPIContext): YDataLogger null
es	static FirstDataLoggerInContext (yctx)

Use the method `YDataLogger.nextDataLogger ( )` to iterate on next data loggers.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YDataLogger` object, corresponding to the first data logger currently online, or a `null` pointer if there are none.

YDataLogger.GetSimilarFunctions()**YDataLogger****YDataLogger.GetSimilarFunctions()**

Enumerates all functions of type DataLogger available on the devices currently reachable by the library, and returns their unique hardware ID.

dnp	<code>static new string[] GetSimilarFunctions()</code>

cp	<code>static vector<string> GetSimilarFunctions()</code>

Each of these IDs can be provided as argument to the method `YDataLogger.FindDataLogger` to obtain an object that can control the corresponding device.

Returns :

an array of strings, each string containing the unique hardwareId of a device function currently connected.

datalogger→AdvertisedValue**YDataLogger**

Short string representing the current state of the function.

dnf [string AdvertisedValue](#)

datalogger→AutoStart**YDataLogger**

Default activation state of the data logger on power up.

dnp `int AutoStart`

Writable. Do not forget to call the `saveToFlash()` method of the module to save the configuration change. Note: if the device doesn't have any time source at his disposal when starting up, it will wait for ~8 seconds before automatically starting to record with an arbitrary timestamp

datalogger→BeaconDriven**YDataLogger**

True if the data logger is synchronised with the localization beacon.

dnf **int BeaconDriven**

Writable. Changes the type of synchronisation of the data logger. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

datalogger→FriendlyName**YDataLogger**

Global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

dnf `string` **FriendlyName**

The returned string uses the logical names of the module and of the function if they are defined, otherwise the serial number of the module and the hardware identifier of the function (for example: `MyCustomName.relay1`)

datalogger→FunctionId**YDataLogger**

Hardware identifier of the data logger, without reference to the module.

dnf `string` **FunctionId**

For example `re lay1`

datalogger→**HardwareId****YDataLogger**

Unique hardware identifier of the function in the form SERIAL . FUNCTIONID.

dnp `string` **HardwareId**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example RELAYL01-123456 . r e l a y 1).

datalogger→IsOnline**YDataLogger**

Checks if the function is currently reachable.

dnf **bool IsOnline**

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the function.

datalogger→LogicalName**YDataLogger**

Logical name of the function.

dnf `string LogicalName`

Writable. You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

datalogger→Recording**YDataLogger**

Current activation state of the data logger.

dnf **int Recording**

Writable. Changes the activation state of the data logger to start/stop recording data.

datalogger→**SerialNumber****YDataLogger**

Serial number of the module, as set by the factory.

dnf string **SerialNumber**

datalogger→clearCache()**YDataLogger**

Invalidates the cache.

js	function clearCache ()
cpp	void clearCache ()
m	-(void) clearCache
pas	clearCache ()
vb	procedure clearCache ()
cs	void clearCache ()
java	void clearCache ()
py	clearCache ()
php	function clearCache ()
ts	async clearCache (): Promise<void>
es	async clearCache ()

Invalidates the cache of the data logger attributes. Forces the next call to `get_xxx()` or `loadxxx()` to use values that come from the device.

datalogger→describe()**YDataLogger**

Returns a short text that describes unambiguously the instance of the data logger in the form
 TYPE(NAME)=SERIAL.FUNCTIONID.

js	function describe ()
cpp	string describe ()
m	-(NSString*) describe
pas	string describe (): string
vb	function describe () As String
cs	string describe ()
java	String describe ()
py	describe ()
php	function describe ()
ts	async describe (): Promise<string>
es	async describe ()

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYL01-123456.relay1 if the module is already connected or Relay(BadCustomName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :

a string that describes the data logger (ex:
 Relay(MyCustomName.relay1)=RELAYL01-123456.relay1)

datalogger→forgetAllDataStreams()**YDataLogger**

Clears the data logger memory and discards all recorded data streams.

js	function forgetAllDataStreams ()
cpp	int forgetAllDataStreams ()
m	-(int) forgetAllDataStreams
pas	LongInt forgetAllDataStreams (): LongInt
vb	function forgetAllDataStreams () As Integer
cs	int forgetAllDataStreams ()
java	int forgetAllDataStreams ()
uwp	async Task<int> forgetAllDataStreams ()
py	forgetAllDataStreams ()
php	function forgetAllDataStreams ()
ts	async forgetAllDataStreams (): Promise<number>
es	async forgetAllDataStreams ()
dnp	int forgetAllDataStreams ()
cp	int forgetAllDataStreams ()
cmd	YDataLogger target forgetAllDataStreams

This method also resets the current run index to zero.

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→**get_advertisedValue()****YDataLogger****datalogger**→**advertisedValue()**

Returns the current value of the data logger (no more than 6 characters).

js	function get_advertisedValue ()
cpp	string get_advertisedValue ()
m	-(NSString*) advertisedValue
pas	string get_advertisedValue (): string
vb	function get_advertisedValue () As String
cs	string get_advertisedValue ()
java	String get_advertisedValue ()
uwp	async Task<string> get_advertisedValue ()
py	get_advertisedValue ()
php	function get_advertisedValue ()
ts	async get_advertisedValue (): Promise<string>
es	async get_advertisedValue ()
dnp	string get_advertisedValue ()
cp	string get_advertisedValue ()
cmd	YDataLogger target get_advertisedValue

Returns :

a string corresponding to the current value of the data logger (no more than 6 characters).

On failure, throws an exception or returns YDataLogger .ADVERTISEDVALUE_INVALID.

datalogger→get_autoStart()**YDataLogger****datalogger→autoStart()**

Returns the default activation state of the data logger on power up.

js	function get_autoStart ()
cpp	Y_AUTOSTART_enum get_autoStart ()
m	-(Y_AUTOSTART_enum) autoStart
pas	Integer get_autoStart (): Integer
vb	function get_autoStart () As Integer
cs	int get_autoStart ()
java	int get_autoStart ()
uwp	async Task<int> get_autoStart ()
py	get_autoStart ()
php	function get_autoStart ()
ts	async get_autoStart (): Promise<YDataLogger_AutoStart>
es	async get_autoStart ()
dnp	int get_autoStart ()
cp	int get_autoStart ()
cmd	YDataLogger target get_autoStart

Returns :

either YDataLogger.AUTOSTART_OFF or YDataLogger.AUTOSTART_ON, according to the default activation state of the data logger on power up

On failure, throws an exception or returns YDataLogger.AUTOSTART_INVALID.

datalogger→get_beaconDriven()**YDataLogger****datalogger→beaconDriven()**

Returns true if the data logger is synchronised with the localization beacon.

js	function get_beaconDriven ()
cpp	Y_BEACONDRIVEN_enum get_beaconDriven ()
m	-(Y_BEACONDRIVEN_enum) beaconDriven
pas	Integer get_beaconDriven (): Integer
vb	function get_beaconDriven () As Integer
cs	int get_beaconDriven ()
java	int get_beaconDriven ()
uwp	async Task<int> get_beaconDriven ()
py	get_beaconDriven ()
php	function get_beaconDriven ()
ts	async get_beaconDriven (): Promise<YDataLogger_BeaconDriven>
es	async get_beaconDriven ()
dnp	int get_beaconDriven ()
cp	int get_beaconDriven ()
cmd	YDataLogger target get_beaconDriven

Returns :

either YDataLogger.BEACONDRIVEN_OFF or YDataLogger.BEACONDRIVEN_ON,
according to true if the data logger is synchronised with the localization beacon

On failure, throws an exception or returns YDataLogger.BEACONDRIVEN_INVALID.

datalogger→get_currentRunIndex()**YDataLogger****datalogger→currentRunIndex()**

Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.

js	function get_currentRunIndex ()
cpp	int get_currentRunIndex ()
m	-(int) currentRunIndex
pas	LongInt get_currentRunIndex (): LongInt
vb	function get_currentRunIndex () As Integer
cs	int get_currentRunIndex ()
java	int get_currentRunIndex ()
uwp	async Task<int> get_currentRunIndex ()
py	get_currentRunIndex ()
php	function get_currentRunIndex ()
ts	async get_currentRunIndex (): Promise<number>
es	async get_currentRunIndex ()
dnp	int get_currentRunIndex ()
cp	int get_currentRunIndex ()
cmd	YDataLogger target get_currentRunIndex

Returns :

an integer corresponding to the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point

On failure, throws an exception or returns YDataLogger.CURRENTRUNINDEX_INVALID.

datalogger→get_dataSets()**YDataLogger****datalogger→dataSets()**

Returns a list of YDataSet objects that can be used to retrieve all measures stored by the data logger.

js	function get_dataSets ()
cpp	vector<YDataSet> get_dataSets ()
m	-(NSMutableArray*) dataSets
pas	TYDataSetArray get_dataSets (): TYDataSetArray
vb	function get_dataSets () As List
cs	List<YDataSet> get_dataSets ()
java	ArrayList<YDataSet> get_dataSets ()
uwp	async Task<List<YDataSet>> get_dataSets ()
py	get_dataSets ()
php	function get_dataSets ()
ts	async get_dataSets (): Promise<YDataSet[]
es	async get_dataSets ()
dnp	YDataSetProxy[] get_dataSets ()
cmd	YDataLogger target get_dataSets

This function only works if the device uses a recent firmware, as YDataSet objects are not supported by firmwares older than version 13000.

Returns :

a list of YDataSet object.

On failure, throws an exception or returns an empty list.

datalogger→get_dataStreams()**YDataLogger****datalogger→dataStreams()**

Builds a list of all data streams hold by the data logger (legacy method).

The caller must pass by reference an empty array to hold `DataStream` objects, and the function fills it with objects describing available data sequences.

This is the old way to retrieve data from the `DataLogger`. For new applications, you should rather use `get_dataSets()` method, or call directly `get_recordedData()` on the sensor object.

Parameters :

▼ an array of `DataStream` objects to be filled in

Returns :

`YAPI . SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→**get_errorMessage()****YDataLogger****datalogger**→**errorMessage()**

Returns the error message of the latest error with the data logger.

js	function get_errorMessage ()
cpp	string get_errorMessage ()
m	-(NSString*) errorMessage
pas	string get_errorMessage (): string
vb	function get_errorMessage () As String
cs	string get_errorMessage ()
java	String get_errorMessage ()
py	get_errorMessage ()
php	function get_errorMessage ()
ts	get_errorMessage (): string
es	get_errorMessage ()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a string corresponding to the latest error message that occurred while using the data logger object

datalogger→get_errorType()**YDataLogger****datalogger→errorType()**

Returns the numerical error code of the latest error with the data logger.

js	function get_errorType ()
cpp	YRETCODE get_errorType ()
m	-(YRETCODE) errorType
pas	YRETCODE get_errorType (): YRETCODE
vb	function get_errorType () As YRETCODE
cs	YRETCODE get_errorType ()
java	int get_errorType ()
py	get_errorType ()
php	function get_errorType ()
ts	get_errorType (): number
es	get_errorType ()

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :

a number corresponding to the code of the latest error that occurred while using the data logger object

datalogger→get_friendlyName()**YDataLogger****datalogger→friendlyName()**

Returns a global identifier of the data logger in the format `MODULE_NAME.FUNCTION_NAME`.

js	function get_friendlyName ()
cpp	string get_friendlyName ()
m	-(NSString*) friendlyName
cs	string get_friendlyName ()
java	String get_friendlyName ()
py	get_friendlyName ()
php	function get_friendlyName ()
ts	async get_friendlyName (): Promise<string>
es	async get_friendlyName ()
dnp	string get_friendlyName ()
cp	string get_friendlyName ()

The returned string uses the logical names of the module and of the data logger if they are defined, otherwise the serial number of the module and the hardware identifier of the data logger (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the data logger using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `YDataLogger.FRIENDLYNAME_INVALID`.

datalogger→get_functionDescriptor()**YDataLogger****datalogger→functionDescriptor()**

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

js	function get_functionDescriptor ()
cpp	YFUN_DESCR get_functionDescriptor ()
m	-(YFUN_DESCR) functionDescriptor
pas	YFUN_DESCR get_functionDescriptor (): YFUN_DESCR
vb	function get_functionDescriptor () As YFUN_DESCR
cs	YFUN_DESCR get_functionDescriptor ()
java	String get_functionDescriptor ()
py	get_functionDescriptor ()
php	function get_functionDescriptor ()
ts	async get_functionDescriptor (): Promise<string>
es	async get_functionDescriptor ()

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y\$CLASSNAME\$.FUNCTIONDESCRIPTOR_INVALID.

datalogger→get_functionId()

YDataLogger

datalogger→functionId()

Returns the hardware identifier of the data logger, without reference to the module.

js	function get_functionId ()
cpp	string get_functionId ()
m	-(NSString*) functionId
vb	function get_functionId () As String
cs	string get_functionId ()
java	String get_functionId ()
py	get_functionId ()
php	function get_functionId ()
ts	async get_functionId (): Promise<string>
es	async get_functionId ()
dnp	string get_functionId ()
cp	string get_functionId ()

For example `relay1`

Returns :

a string that identifies the data logger (ex: `relay1`)

On failure, throws an exception or returns `YDataLogger.FUNCTIONID_INVALID`.

datalogger→get_hardwareId()**YDataLogger****datalogger→hardwareId()**

Returns the unique hardware identifier of the data logger in the form SERIAL . FUNCTIONID.

js	function get_hardwareId ()
cpp	string get_hardwareId ()
m	-(NSString*) hardwareId
vb	function get_hardwareId () As String
cs	string get_hardwareId ()
java	String get_hardwareId ()
py	get_hardwareId ()
php	function get_hardwareId ()
ts	async get_hardwareId (): Promise<string>
es	async get_hardwareId ()
dnp	string get_hardwareId ()
cp	string get_hardwareId ()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the data logger (for example RELAYL01-123456 . re lay1).

Returns :

a string that uniquely identifies the data logger (ex: RELAYL01-123456 . re lay1)

On failure, throws an exception or returns YDataLogger . HARDWAREID_INVALID.

datalogger→get_logicalName()**YDataLogger****datalogger→logicalName()**

Returns the logical name of the data logger.

js	function get_logicalName ()
cpp	string get_logicalName ()
m	-(NSString*) logicalName
pas	string get_logicalName (): string
vb	function get_logicalName () As String
cs	string get_logicalName ()
java	String get_logicalName ()
uwp	async Task<string> get_logicalName ()
py	get_logicalName ()
php	function get_logicalName ()
ts	async get_logicalName (): Promise<string>
es	async get_logicalName ()
dnp	string get_logicalName ()
cp	string get_logicalName ()
cmd	YDataLogger target get_logicalName

Returns :

a string corresponding to the logical name of the data logger.

On failure, throws an exception or returns YDataLogger.LOGICALNAME_INVALID.

datalogger→get_module()**YDataLogger****datalogger→module()**

Gets the YModule object for the device on which the function is located.

js	function get_module ()
cpp	YModule * get_module ()
m	-(YModule*) module
pas	TYModule get_module (): TYModule
vb	function get_module () As YModule
cs	YModule get_module ()
java	YModule get_module ()
py	get_module ()
php	function get_module ()
ts	async get_module (): Promise<YModule>
es	async get_module ()
dnp	YModuleProxy get_module ()
cp	YModuleProxy * get_module ()

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :

an instance of YModule

datalogger→**get_module_async()****YDataLogger****datalogger**→**module_async()**

Gets the YModule object for the device on which the function is located (asynchronous version).

```
js function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as on-line.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking Firefox JavaScript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous JavaScript calls for more details.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

datalogger→get_recording()**YDataLogger****datalogger→recording()**

Returns the current activation state of the data logger.

js	function get_recording ()
cpp	Y_RECORDING_enum get_recording ()
m	-(Y_RECORDING_enum) recording
pas	Integer get_recording (): Integer
vb	function get_recording () As Integer
cs	int get_recording ()
java	int get_recording ()
uwp	async Task<int> get_recording ()
py	get_recording ()
php	function get_recording ()
ts	async get_recording (): Promise<YDataLogger_Recording>
es	async get_recording ()
dnp	int get_recording ()
cp	int get_recording ()
cmd	YDataLogger target get_recording

Returns :

a value among `YDataLogger.RECORDING_OFF`, `YDataLogger.RECORDING_ON` and `YDataLogger.RECORDING_PENDING` corresponding to the current activation state of the data logger

On failure, throws an exception or returns `YDataLogger.RECORDING_INVALID`.

datalogger→**get_serialNumber()****YDataLogger****datalogger**→**serialNumber()**

Returns the serial number of the module, as set by the factory.

js	function get_serialNumber ()
cpp	string get_serialNumber ()
m	-(NSString*) serialNumber
pas	string get_serialNumber (): string
vb	function get_serialNumber () As String
cs	string get_serialNumber ()
java	String get_serialNumber ()
uwp	async Task<string> get_serialNumber ()
py	get_serialNumber ()
php	function get_serialNumber ()
ts	async get_serialNumber (): Promise<string>
es	async get_serialNumber ()
dnp	string get_serialNumber ()
cp	string get_serialNumber ()
cmd	YDataLogger target get_serialNumber

Returns :

a string corresponding to the serial number of the module, as set by the factory.

On failure, throws an exception or returns YFunction.SERIALNUMBER_INVALID.

datalogger→get_timeUTC()**YDataLogger****datalogger→timeUTC()**

Returns the Unix timestamp for current UTC time, if known.

js	function get_timeUTC ()
cpp	s64 get_timeUTC ()
m	-(s64) timeUTC
pas	int64 get_timeUTC (): int64
vb	function get_timeUTC () As Long
cs	long get_timeUTC ()
java	long get_timeUTC ()
uwp	async Task<long> get_timeUTC ()
py	get_timeUTC ()
php	function get_timeUTC ()
ts	async get_timeUTC (): Promise<number>
es	async get_timeUTC ()
dnp	long get_timeUTC ()
cp	s64 get_timeUTC ()
cmd	YDataLogger target get_timeUTC

Returns :

an integer corresponding to the Unix timestamp for current UTC time, if known

On failure, throws an exception or returns YDataLogger . TIMEUTC_INVALID.

datalogger→get_usage()**YDataLogger****datalogger→usage()**

Returns the percentage of datalogger memory in use.

js	function get_usage ()
cpp	int get_usage ()
m	-(int) usage
pas	LongInt get_usage (): LongInt
vb	function get_usage () As Integer
cs	int get_usage ()
java	int get_usage ()
uwp	async Task<int> get_usage ()
py	get_usage ()
php	function get_usage ()
ts	async get_usage (): Promise<number>
es	async get_usage ()
dnp	int get_usage ()
cp	int get_usage ()
cmd	YDataLogger target get_usage

Returns :

an integer corresponding to the percentage of datalogger memory in use

On failure, throws an exception or returns YDataLogger .USAGE_INVALID.

datalogger→get_userdata()**YDataLogger****datalogger→userdata()**

Returns the value of the `userData` attribute, as previously stored using method `set_userdata`.

js	function get_userdata ()
cpp	void * get_userdata ()
m	-(id) <code>userData</code>
pas	Tobject get_userdata (): Tobject
vb	function get_userdata () As Object
cs	object get_userdata ()
java	Object get_userdata ()
py	get_userdata ()
php	function get_userdata ()
ts	async get_userdata (): Promise<object null>
es	async get_userdata ()

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :

the object stored previously by the caller.

datalogger→**isOnline()****YDataLogger**

Checks if the data logger is currently reachable, without raising any error.

js	function isOnline ()
cpp	bool isOnline ()
m	-(BOOL) isOnline
pas	boolean isOnline (): boolean
vb	function isOnline () As Boolean
cs	bool isOnline ()
java	boolean isOnline ()
py	isOnline ()
php	function isOnline ()
ts	async isOnline (): Promise<boolean>
es	async isOnline ()
dnp	bool isOnline ()
cp	bool isOnline ()

If there is a cached value for the data logger in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the data logger.

Returns :

`true` if the data logger can be reached, and `false` otherwise

datalogger→isOnline_async()**YDataLogger**

Checks if the data logger is currently reachable, without raising any error (asynchronous version).

```
js function isOnline_async( callback, context)
```

If there is a cached value for the data logger in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

datalogger→**isReadOnly()****YDataLogger**

Test if the function is readOnly.

cpp	<code>bool isReadOnly()</code>
m	<code>-(bool) isReadOnly</code>
pas	<code>boolean isReadOnly(): boolean</code>
vb	<code>function isReadOnly() As Boolean</code>
cs	<code>bool isReadOnly()</code>
java	<code>boolean isReadOnly()</code>
uwp	<code>async Task<bool> isReadOnly()</code>
py	<code>isReadOnly()</code>
php	<code>function isReadOnly()</code>
ts	<code>async isReadOnly(): Promise<boolean></code>
es	<code>async isReadOnly()</code>
dnp	<code>bool isReadOnly()</code>
cp	<code>bool isReadOnly()</code>
cmd	<code>YDataLogger target isReadOnly</code>

Return `true` if the function is write protected or that the function is not available.

Returns :

`true` if the function is readOnly or not online.

datalogger→load()**YDataLogger**

Preloads the data logger cache with a specified validity duration.

js	function load (msValidity)
cpp	YRETCODE load (int msValidity)
m	-(YRETCODE) load : (u64) msValidity
pas	YRETCODE load (msValidity : u64): YRETCODE
vb	function load (ByVal msValidity As Long) As YRETCODE
cs	YRETCODE load (ulong msValidity)
java	int load (long msValidity)
py	load (msValidity)
php	function load (\$msValidity)
ts	async load (msValidity : number): Promise<number>
es	async load (msValidity)

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI . SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→loadAttribute()**YDataLogger**

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

js	function loadAttribute (attrName)
cpp	string loadAttribute (string attrName)
m	-(NSString*) loadAttribute : (NSString*) attrName
pas	string loadAttribute (attrName : string): string
vb	function loadAttribute (ByVal attrName As String) As String
cs	string loadAttribute (string attrName)
java	String loadAttribute (String attrName)
uwp	async Task<string> loadAttribute (string attrName)
py	loadAttribute (attrName)
php	function loadAttribute (\$ attrName)
ts	async loadAttribute (attrName : string): Promise<string>
es	async loadAttribute (attrName)
dnp	string loadAttribute (string attrName)
cp	string loadAttribute (string attrName)

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

datalogger→load_async()**YDataLogger**

Preloads the data logger cache with a specified validity duration (asynchronous version).

```
js function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking the JavaScript virtual machine.

Parameters :

- msValidity** an integer corresponding to the validity of the loaded function parameters, in milliseconds
- callback** callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or YAPI . SUCCESS)
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

datalogger→muteValueCallbacks()**YDataLogger**

Disables the propagation of every new advertised value to the parent hub.

js	function muteValueCallbacks ()
cpp	int muteValueCallbacks ()
m	-(int) muteValueCallbacks
pas	LongInt muteValueCallbacks (): LongInt
vb	function muteValueCallbacks () As Integer
cs	int muteValueCallbacks ()
java	int muteValueCallbacks ()
uwp	async Task<int> muteValueCallbacks ()
py	muteValueCallbacks ()
php	function muteValueCallbacks ()
ts	async muteValueCallbacks (): Promise<number>
es	async muteValueCallbacks ()
dnp	int muteValueCallbacks ()
cp	int muteValueCallbacks ()
cmd	YDataLogger target muteValueCallbacks

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

YAPI . SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→nextDataLogger()**YDataLogger**

Continues the enumeration of data loggers started using `yFirstDataLogger()`.

js	<code>function nextDataLogger()</code>
cpp	<code>YDataLogger * nextDataLogger()</code>
m	<code>-(nullable YDataLogger*) nextDataLogger</code>
pas	<code>TYDataLogger nextDataLogger(): TYDataLogger</code>
vb	<code>function nextDataLogger() As YDataLogger</code>
cs	<code>YDataLogger nextDataLogger()</code>
java	<code>YDataLogger nextDataLogger()</code>
uwp	<code>YDataLogger nextDataLogger()</code>
py	<code>nextDataLogger()</code>
php	<code>function nextDataLogger()</code>
ts	<code>nextDataLogger(): YDataLogger null</code>
es	<code>nextDataLogger()</code>

Caution: You can't make any assumption about the returned data loggers order. If you want to find a specific a data logger, use `DataLogger.findDataLogger()` and a `hardwareID` or a logical name.

Returns :

a pointer to a `YDataLogger` object, corresponding to a data logger currently online, or a `null` pointer if there are no more data loggers to enumerate.

datalogger→registerValueCallback()**YDataLogger**

Registers the callback function that is invoked on every change of advertised value.

js	function registerValueCallback (callback)
cpp	int registerValueCallback (YDataLoggerValueCallback callback)
m	-(int) registerValueCallback : (YDataLoggerValueCallback _Nullable) callback
pas	LongInt registerValueCallback (callback : TYDataLoggerValueCallback): LongInt
vb	function registerValueCallback (ByVal callback As YDataLoggerValueCallback) As Integer
cs	int registerValueCallback (ValueCallback callback)
java	int registerValueCallback (UpdateCallback callback)
uwp	async Task<int> registerValueCallback (ValueCallback callback)
py	registerValueCallback (callback)
php	function registerValueCallback (\$callback)
ts	async registerValueCallback (callback : YDataLoggerValueCallback null): Promise<number>
es	async registerValueCallback (callback)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

datalogger→set_autoStart()**YDataLogger****datalogger→setAutoStart()**

Changes the default activation state of the data logger on power up.

js	function set_autoStart (newval)
cpp	int set_autoStart (Y_AUTOSTART_enum newval)
m	-(int) setAutoStart : (Y_AUTOSTART_enum) newval
pas	integer set_autoStart (newval : Integer): integer
vb	function set_autoStart (ByVal newval As Integer) As Integer
cs	int set_autoStart (int newval)
java	int set_autoStart (int newval)
uwp	async Task<int> set_autoStart (int newval)
py	set_autoStart (newval)
php	function set_autoStart (\$newval)
ts	async set_autoStart (newval : YDataLogger_AutoStart): Promise<number>
es	async set_autoStart (newval)
dnp	int set_autoStart (int newval)
cp	int set_autoStart (int newval)
cmd	YDataLogger target set_autoStart newval

Do not forget to call the `saveToFlash( )` method of the module to save the configuration change.
 Note: if the device doesn't have any time source at his disposal when starting up, it will wait for ~8 seconds before automatically starting to record with an arbitrary timestamp

Parameters :

newval either `YDataLogger.AUTOSTART_OFF` or `YDataLogger.AUTOSTART_ON`, according to the default activation state of the data logger on power up

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→set_beaconDriven()**YDataLogger****datalogger→setBeaconDriven()**

Changes the type of synchronisation of the data logger.

js	function set_beaconDriven (newval)
cpp	int set_beaconDriven (Y_BEACONDRIVEN_enum newval)
m	-(int) setBeaconDriven : (Y_BEACONDRIVEN_enum) newval
pas	integer set_beaconDriven (newval : Integer): integer
vb	function set_beaconDriven (ByVal newval As Integer) As Integer
cs	int set_beaconDriven (int newval)
java	int set_beaconDriven (int newval)
uwp	async Task<int> set_beaconDriven (int newval)
py	set_beaconDriven (newval)
php	function set_beaconDriven (\$ newval)
ts	async set_beaconDriven (newval : YDataLogger_BeaconDriven): Promise<number>
es	async set_beaconDriven (newval)
dnp	int set_beaconDriven (int newval)
cp	int set_beaconDriven (int newval)
cmd	YDataLogger target set_beaconDriven newval

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval either `YDataLogger.BEACONDRIVEN_OFF` or `YDataLogger.BEACONDRIVEN_ON`, according to the type of synchronisation of the data logger

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→set_logicalName()**YDataLogger****datalogger→setLogicalName()**

Changes the logical name of the data logger.

js	function set_logicalName (newval)
cpp	int set_logicalName (string newval)
m	-(int) setLogicalName : (NSString*) newval
pas	integer set_logicalName (newval : string): integer
vb	function set_logicalName (ByVal newval As String) As Integer
cs	int set_logicalName (string newval)
java	int set_logicalName (String newval)
uwp	async Task<int> set_logicalName (string newval)
py	set_logicalName (newval)
php	function set_logicalName (\$ newval)
ts	async set_logicalName (newval : string): Promise<number>
es	async set_logicalName (newval)
dnp	int set_logicalName (string newval)
cp	int set_logicalName (string newval)
cmd	YDataLogger target set_logicalName newval

You can use `yCheckLogicalName( )` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash( )` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the data logger.

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→set_recording()**YDataLogger****datalogger→setRecording()**

Changes the activation state of the data logger to start/stop recording data.

js	function set_recording (newval)
cpp	int set_recording (Y_RECORDING_enum newval)
m	-(int) setRecording : (Y_RECORDING_enum) newval
pas	integer set_recording (newval : Integer): integer
vb	function set_recording (ByVal newval As Integer) As Integer
cs	int set_recording (int newval)
java	int set_recording (int newval)
uwp	async Task<int> set_recording (int newval)
py	set_recording (newval)
php	function set_recording (\$newval)
ts	async set_recording (newval : YDataLogger_Recording): Promise<number>
es	async set_recording (newval)
dnp	int set_recording (int newval)
cp	int set_recording (int newval)
cmd	YDataLogger target set_recording newval

Parameters :

newval a value among YDataLogger.RECORDING_OFF, YDataLogger.RECORDING_ON and YDataLogger.RECORDING_PENDING corresponding to the activation state of the data logger to start/stop recording data

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→set_timeUTC()**YDataLogger****datalogger→setTimeUTC()**

Changes the current UTC time reference used for recorded data.

js	function set_timeUTC (newval)
cpp	int set_timeUTC (s64 newval)
m	-(int) setTimeUTC : (s64) newval
pas	integer set_timeUTC (newval : int64): integer
vb	function set_timeUTC (ByVal newval As Long) As Integer
cs	int set_timeUTC (long newval)
java	int set_timeUTC (long newval)
uwp	async Task<int> set_timeUTC (long newval)
py	set_timeUTC (newval)
php	function set_timeUTC (\$ newval)
ts	async set_timeUTC (newval : number): Promise<number>
es	async set_timeUTC (newval)
dnp	int set_timeUTC (long newval)
cp	int set_timeUTC (s64 newval)
cmd	YDataLogger target set_timeUTC newval

Parameters :

newval an integer corresponding to the current UTC time reference used for recorded data

Returns :

YAPI . SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→**set_userData()****YDataLogger****datalogger**→**setUserData()**

Stores a user context provided as argument in the `userData` attribute of the function.

js	<code>function set_userData(data)</code>
cpp	<code>void set_userData(void * data)</code>
m	<code>-(void) setUserData : (id) data</code>
pas	<code>set_userData(data: Tobject)</code>
vb	<code>procedure set_userData(ByVal data As Object)</code>
cs	<code>void set_userData(object data)</code>
java	<code>void set_userData(Object data)</code>
py	<code>set_userData(data)</code>
php	<code>function set_userData(\$data)</code>
ts	<code>async set_userData(data: object null): Promise<void></code>
es	<code>async set_userData(data)</code>

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

datalogger→**unmuteValueCallbacks()****YDataLogger**

Re-enables the propagation of every new advertised value to the parent hub.

js	function unmuteValueCallbacks ()
cpp	int unmuteValueCallbacks ()
m	-(int) unmuteValueCallbacks
pas	LongInt unmuteValueCallbacks (): LongInt
vb	function unmuteValueCallbacks () As Integer
cs	int unmuteValueCallbacks ()
java	int unmuteValueCallbacks ()
uwp	async Task<int> unmuteValueCallbacks ()
py	unmuteValueCallbacks ()
php	function unmuteValueCallbacks ()
ts	async unmuteValueCallbacks (): Promise<number>
es	async unmuteValueCallbacks ()
dnp	int unmuteValueCallbacks ()
cp	int unmuteValueCallbacks ()
cmd	YDataLogger target unmuteValueCallbacks

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :

YAPI . SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

datalogger→wait_async()**YDataLogger**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js function wait_async( callback, context)
ts wait_async( callback: Function, context: object)
es wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

Parameters :

- callback** callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.
- context** caller-specific object that is passed as-is to the callback function

Returns :

nothing.

24.7. Class YDataSet

Recorded data sequence, as returned by `sensor.get_recordedData()`

`YDataSet` objects make it possible to retrieve a set of recorded measures for a given sensor and a specified time interval. They can be used to load data points with a progress report. When the `YDataSet` object is instantiated by the `sensor.get_recordedData()` function, no data is yet loaded from the module. It is only when the `loadMore()` method is called over and over than data will be effectively loaded from the `dataLogger`.

A preview of available measures is available using the function `get_preview()` as soon as `loadMore()` has been called once. Measures themselves are available using function `get_measures()` when loaded by subsequent calls to `loadMore()`.

This class can only be used on devices that use a relatively recent firmware, as `YDataSet` objects are not supported by firmwares older than version 13000.

In order to use the functions described here, you should include:

js	<code><script type='text/javascript' src='yocto_module.js'></script></code>
cpp	<code>#include "yocto_module.h"</code>
m	<code>#import "yocto_module.h"</code>
pas	<code>uses yocto_module;</code>
vb	<code>yocto_module.vb</code>
cs	<code>yocto_module.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YDataSet;</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YDataSet;</code>
py	<code>from yocto_module import *</code>
php	<code>require_once('yocto_module.php');</code>
ts	in HTML: <code>import { YDataSet } from '../dist/esm/yocto_module.js';</code> in Node.js: <code>import { YDataSet } from 'yoctolib-cjs/yocto_module.js';</code>
es	in HTML: <code><script src='../lib/yocto_module.js'></script></code> in node.js: <code>require('yoctolib-es2017/yocto_module.js');</code>
dnp	<code>import YoctoProxyAPI.YDataSetProxy</code>
cp	<code>#include "yocto_module_proxy.h"</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.

`dataset→get_functionId()`

Returns the hardware identifier of the function that performed the measure, without reference to the module.

`dataset→get_hardwareId()`

Returns the unique hardware identifier of the function who performed the measures, in the form `SERIAL.FUNCTIONID`.

`dataset→get_measures()`

Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

dataset→get_measuresAt(measure)

Returns the detailed set of measures for the time interval corresponding to a given condensed measures previously returned by get_preview().

dataset→get_measuresAvgAt(index)

Returns the average value observed during the time interval covered by the specified entry in the preview.

dataset→get_measuresEndTimeAt(index)

Returns the end time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

dataset→get_measuresMaxAt(index)

Returns the largest value observed during the time interval covered by the specified entry in the preview.

dataset→get_measuresMinAt(index)

Returns the smallest value observed during the time interval covered by the specified entry in the preview.

dataset→get_measuresRecordCount()

Returns the number of measurements currently loaded for this data set.

dataset→get_measuresStartTimeAt(index)

Returns the start time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

dataset→get_preview()

Returns a condensed version of the measures that can retrieved in this YDataSet, as a list of YMeasure objects.

dataset→get_previewAvgAt(index)

Returns the average value observed during the time interval covered by the specified entry in the preview.

dataset→get_previewEndTimeAt(index)

Returns the end time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

dataset→get_previewMaxAt(index)

Returns the largest value observed during the time interval covered by the specified entry in the preview.

dataset→get_previewMinAt(index)

Returns the smallest value observed during the time interval covered by the specified entry in the preview.

dataset→get_previewRecordCount()

Returns the number of entries in the preview summarizing this data set

dataset→get_previewStartTimeAt(index)

Returns the start time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

dataset→get_progress()

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

dataset→get_startTimeUTC()

Returns the start time of the dataset, relative to the Jan 1, 1970.

dataset→get_summary()

Returns an YMeasure object which summarizes the whole YDataSet.

dataset→get_summaryAvg()

Returns the average value observed during the time interval covered by this data set.

dataset→get_summaryEndTime()

Returns the end time of the last measure in the data set, relative to the Jan 1, 1970 UTC (Unix timestamp).

dataset→get_summaryMax()

Returns the largest value observed during the time interval covered by this data set.

dataset→get_summaryMin()

Returns the smallest value observed during the time interval covered by this data set.

dataset→get_summaryStartTime()

Returns the start time of the first measure in the data set, relative to the Jan 1, 1970 UTC (Unix timestamp).

dataset→get_unit()

Returns the measuring unit for the measured value.

dataset→loadMore()

Loads the the next block of measures from the dataLogger, and updates the progress indicator.

dataset→loadMore_async(callback, context)

Loads the the next block of measures from the dataLogger asynchronously.

YDataSet.Init()**YDataSet****YDataSet.Init()**

Retrieves a `YDataSet` object holding historical data for a sensor given by its name or hardware identifier, for a specified time interval.

The measures will be retrieved from the data logger, which must have been turned on at the desired time. Methods of the `YDataSet` class makes it possible to get an overview of the recorded data, and to load progressively a large set of measures from the data logger.

Parameters :

- sensorName** logical name or hardware identifier of the sensor for which data logger records are requested.
- startTime** the start of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without initial limit.
- endTime** the end of the desired measure time interval, as a Unix timestamp, i.e. the number of seconds since January 1, 1970 UTC. The special value 0 can be used to include any measure, without ending limit.

Returns :

an instance of `YDataSet`, providing access to historical data. Past measures can be loaded progressively using methods from the `YDataSet` object.

dataset→**get_endTimeUTC()****YDataSet****dataset**→**endTimeUTC()**

Returns the end time of the dataset, relative to the Jan 1, 1970.

js	function get_endTimeUTC ()
cpp	s64 get_endTimeUTC ()
m	-(s64) endTimeUTC
pas	int64 get_endTimeUTC (): int64
vb	function get_endTimeUTC () As Long
cs	long get_endTimeUTC ()
java	long get_endTimeUTC ()
uwp	async Task<long> get_endTimeUTC ()
py	get_endTimeUTC ()
php	function get_endTimeUTC ()
ts	async get_endTimeUTC (): Promise<number>
es	async get_endTimeUTC ()
dnp	long get_endTimeUTC ()
cp	s64 get_endTimeUTC ()

When the YDataSet object is created, the end time is the value passed in parameter to the `get_dataSet()` function. After the very first call to `loadMore()`, the end time is updated to reflect the timestamp of the last measure actually found in the dataLogger within the specified range.

DEPRECATED: This method has been replaced by `get_summary()` which contain more precise informations.

Returns :

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the end of this data set (i.e. Unix time representation of the absolute time).

dataset→**get_functionId()****dataset**→**functionId()**

Returns the hardware identifier of the function that performed the measure, without reference to the module.

js	function get_functionId ()
cpp	string get_functionId ()
m	-(NSString*) functionId
pas	string get_functionId (): string
vb	function get_functionId () As String
cs	string get_functionId ()
java	String get_functionId ()
uwp	async Task<string> get_functionId ()
py	get_functionId ()
php	function get_functionId ()
ts	async get_functionId (): Promise<string>
es	async get_functionId ()
dnp	string get_functionId ()
cp	string get_functionId ()

For example temperature1.

Returns :

a string that identifies the function (ex: temperature1)

dataset→get_hardwareId()**YDataSet****dataset→hardwareId()**

Returns the unique hardware identifier of the function who performed the measures, in the form `SERIAL.FUNCTIONID`.

js	function get_hardwareId ()
cpp	string get_hardwareId ()
m	-(NSString*) hardwareId
pas	string get_hardwareId (): string
vb	function get_hardwareId () As String
cs	string get_hardwareId ()
java	String get_hardwareId ()
uwp	async Task<string> get_hardwareId ()
py	get_hardwareId ()
php	function get_hardwareId ()
ts	async get_hardwareId (): Promise<string>
es	async get_hardwareId ()
dnp	string get_hardwareId ()
cp	string get_hardwareId ()

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example `THRMCPL1-123456.temperature1`)

Returns :

a string that uniquely identifies the function (ex: `THRMCPL1-123456.temperature1`)

On failure, throws an exception or returns `YDataSet.HARDWAREID_INVALID`.

dataset→get_measures()**dataset→measures()**

Returns all measured values currently available for this DataSet, as a list of YMeasure objects.

js	function get_measures ()
cpp	vector<YMeasure> get_measures ()
m	-(NSMutableArray*) measures
pas	TYMeasureArray get_measures (): TYMeasureArray
vb	function get_measures () As List
cs	List<YMeasure> get_measures ()
java	ArrayList<YMeasure> get_measures ()
uwp	async Task<List<YMeasure>> get_measures ()
py	get_measures ()
php	function get_measures ()
ts	async get_measures (): Promise<YMeasure[]
es	async get_measures ()
dnp	YMeasure[] get_measures ()
cp	vector<YMeasure> get_measures ()

Each item includes: - the start of the measure time interval - the end of the measure time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

Before calling this method, you should call `LoadMore()` to load data from the device. You may have to call `loadMore()` several time until all rows are loaded, but you can start looking at available data rows before the load is complete.

The oldest measures are always loaded first, and the most recent measures will be loaded last. As a result, timestamps are normally sorted in ascending order within the measure table, unless there was an unexpected adjustment of the datalogger UTC clock.

Returns :

a table of records, where each record depicts the measured value for a given time interval

On failure, throws an exception or returns an empty array.

dataset→**get_measuresAt()****YDataSet****dataset**→**measuresAt()**

Returns the detailed set of measures for the time interval corresponding to a given condensed measures previously returned by `get_preview()`.

js	<code>function get_measuresAt(measure)</code>
cpp	<code>vector<YMeasure> get_measuresAt(YMeasure measure)</code>
m	<code>-(NSMutableArray*) measuresAt : (YMeasure*) measure</code>
pas	<code>TYMeasureArray get_measuresAt(measure: TYMeasure): TYMeasureArray</code>
vb	<code>function get_measuresAt(ByVal measure As YMeasure) As List</code>
cs	<code>List<YMeasure> get_measuresAt(YMeasure measure)</code>
java	<code>ArrayList<YMeasure> get_measuresAt(YMeasure measure)</code>
uwp	<code>async Task<List<YMeasure>> get_measuresAt(YMeasure measure)</code>
py	<code>get_measuresAt(measure)</code>
php	<code>function get_measuresAt(\$measure)</code>
ts	<code>async get_measuresAt(measure: YMeasure): Promise<YMeasure[]</code>
es	<code>async get_measuresAt(measure)</code>
dnp	<code>YMeasure[] get_measuresAt(YMeasure measure)</code>
cp	<code>vector<YMeasure> get_measuresAt(YMeasure measure)</code>

The result is provided as a list of `YMeasure` objects.

Parameters :

measure condensed measure from the list previously returned by `get_preview()`.

Returns :

a table of records, where each record depicts the measured values during a time interval

On failure, throws an exception or returns an empty array.

dataset→get_measuresAvgAt()**YDataSet****dataset→measuresAvgAt()**

Returns the average value observed during the time interval covered by the specified entry in the preview.

Parameters :

index an integer index in the range [0...MeasuresRecordCount-1].

Returns :

a floating-point number corresponding to the average value observed.

dataset→get_measuresEndTimeAt()**YDataSet****dataset→measuresEndTimeAt()**

Returns the end time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

When the recording rate is higher than 1 sample per second, the timestamp may have a fractional part.

Parameters :

index an integer index in the range [0...MeasuresRecordCount-1].

Returns :

a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

dataset→get_measuresMaxAt()**YDataSet****dataset→measuresMaxAt()**

Returns the largest value observed during the time interval covered by the specified entry in the preview.

Parameters :

index an integer index in the range [0...MeasuresRecordCount-1].

Returns :

a floating-point number corresponding to the largest value observed.

dataset→get_measuresMinAt()**YDataSet****dataset→measuresMinAt()**

Returns the smallest value observed during the time interval covered by the specified entry in the preview.

Parameters :

index an integer index in the range [0...MeasuresRecordCount-1].

Returns :

a floating-point number corresponding to the smallest value observed.

dataset→get_measuresRecordCount()**YDataSet****dataset→measuresRecordCount()**

Returns the number of measurements currently loaded for this data set.

The total number of record is only known when the data set is fully loaded, i.e. when `loadMore()` has been invoked until the progress indicator returns 100.

Returns :

an integer number corresponding to the number of entries loaded.

dataset→get_measuresStartTimeAt()**YDataSet****dataset→measuresStartTimeAt()**

Returns the start time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

When the recording rate is higher than 1 sample per second, the timestamp may have a fractional part.

Parameters :

index an integer index in the range [0...MeasuresRecordCount-1].

Returns :

a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

dataset→get_preview()**YDataSet****dataset→preview()**

Returns a condensed version of the measures that can be retrieved in this YDataSet, as a list of YMeasure objects.

js	function get_preview ()
cpp	vector<YMeasure> get_preview ()
m	-(NSMutableArray*) preview
pas	TYMeasureArray get_preview (): TYMeasureArray
vb	function get_preview () As List
cs	List<YMeasure> get_preview ()
java	ArrayList<YMeasure> get_preview ()
uwp	async Task<List<YMeasure>> get_preview ()
py	get_preview ()
php	function get_preview ()
ts	async get_preview (): Promise<YMeasure[]
es	async get_preview ()
dnp	YMeasure[] get_preview ()
cp	vector<YMeasure> get_preview ()

Each item includes: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This preview is available as soon as `loadMore()` has been called for the first time.

Returns :

a table of records, where each record depicts the measured values during a time interval

On failure, throws an exception or returns an empty array.

dataset→get_previewAvgAt()**YDataSet****dataset→previewAvgAt()**

Returns the average value observed during the time interval covered by the specified entry in the preview.

Parameters :

index an integer index in the range [0...PreviewRecordCount-1].

Returns :

a floating-point number corresponding to the average value observed.

dataset→get_previewEndTimeAt()**YDataSet****dataset→previewEndTimeAt()**

Returns the end time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

When the recording rate is higher than 1 sample per second, the timestamp may have a fractional part.

Parameters :

index an integer index in the range [0...PreviewRecordCount-1].

Returns :

a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

dataset→**get_previewMaxAt()****YDataSet****dataset**→**previewMaxAt()**

Returns the largest value observed during the time interval covered by the specified entry in the preview.

Parameters :

index an integer index in the range [0...PreviewRecordCount-1].

Returns :

a floating-point number corresponding to the largest value observed.

dataset→get_previewMinAt()**YDataSet****dataset→previewMinAt()**

Returns the smallest value observed during the time interval covered by the specified entry in the preview.

Parameters :

index an integer index in the range [0...PreviewRecordCount-1].

Returns :

a floating-point number corresponding to the smallest value observed.

dataset→get_previewRecordCount()**YDataSet****dataset→previewRecordCount()**

Returns the number of entries in the preview summarizing this data set

Returns :

an integer number corresponding to the number of entries.

dataset→**get_previewStartTimeAt()****YDataSet****dataset**→**previewStartTimeAt()**

Returns the start time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

When the recording rate is higher than 1 sample per second, the timestamp may have a fractional part.

Parameters :

index an integer index in the range [0...PreviewRecordCount-1].

Returns :

a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

dataset→get_progress()**YDataSet****dataset→progress()**

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

js	function get_progress ()
cpp	int get_progress ()
m	-(int) progress
pas	LongInt get_progress (): LongInt
vb	function get_progress () As Integer
cs	int get_progress ()
java	int get_progress ()
uwp	async Task<int> get_progress ()
py	get_progress ()
php	function get_progress ()
ts	async get_progress (): Promise<number>
es	async get_progress ()
dnp	int get_progress ()
cp	int get_progress ()

When the object is instantiated by `get_dataSet`, the progress is zero. Each time `loadMore( )` is invoked, the progress is updated, to reach the value 100 only once all measures have been loaded.

Returns :

an integer in the range 0 to 100 (percentage of completion).

dataset→**get_startTimeUTC()****YDataSet****dataset**→**startTimeUTC()**

Returns the start time of the dataset, relative to the Jan 1, 1970.

js	function get_startTimeUTC ()
cpp	s64 get_startTimeUTC ()
m	-(s64) startTimeUTC
pas	int64 get_startTimeUTC (): int64
vb	function get_startTimeUTC () As Long
cs	long get_startTimeUTC ()
java	long get_startTimeUTC ()
uwp	async Task<long> get_startTimeUTC ()
py	get_startTimeUTC ()
php	function get_startTimeUTC ()
ts	async get_startTimeUTC (): Promise<number>
es	async get_startTimeUTC ()
dnp	long get_startTimeUTC ()
cp	s64 get_startTimeUTC ()

When the YDataSet object is created, the start time is the value passed in parameter to the `get_dataSet( )` function. After the very first call to `loadMore( )`, the start time is updated to reflect the timestamp of the first measure actually found in the dataLogger within the specified range.

DEPRECATED: This method has been replaced by `get_summary( )` which contain more precise informations.

Returns :

an unsigned number corresponding to the number of seconds between the Jan 1, 1970 and the beginning of this data set (i.e. Unix time representation of the absolute time).

dataset→**get_summary()****YDataSet****dataset**→**summary()**

Returns an YMeasure object which summarizes the whole YDataSet.

js	function get_summary ()
cpp	YMeasure get_summary ()
m	-(YMeasure*) summary
pas	TYMeasure get_summary (): TYMeasure
vb	function get_summary () As YMeasure
cs	YMeasure get_summary ()
java	YMeasure get_summary ()
uwp	async Task<YMeasure> get_summary ()
py	get_summary ()
php	function get_summary ()
ts	async get_summary (): Promise<YMeasure>
es	async get_summary ()
dnp	YMeasure get_summary ()
cp	YMeasure get_summary ()

It includes the following information: - the start of a time interval - the end of a time interval - the minimal value observed during the time interval - the average value observed during the time interval - the maximal value observed during the time interval

This summary is available as soon as `LoadMore( )` has been called for the first time.

Returns :

an YMeasure object

dataset→**get_summaryAvg()****YDataSet****dataset**→**summaryAvg()**

Returns the average value observed during the time interval covered by this data set.

Returns :

a floating-point number corresponding to the average value observed.

dataset→get_summaryEndTime()**YDataSet****dataset→summaryEndTime()**

Returns the end time of the last measure in the data set, relative to the Jan 1, 1970 UTC (Unix timestamp).

When the recording rate is higher than 1 sample per second, the timestamp may have a fractional part.

Returns :

a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

dataset→**get_summaryMax()****YDataSet****dataset**→**summaryMax()**

Returns the largest value observed during the time interval covered by this data set.

Returns :

a floating-point number corresponding to the largest value observed.

dataset→get_summaryMin()**YDataSet****dataset→summaryMin()**

Returns the smallest value observed during the time interval covered by this data set.

Returns :

a floating-point number corresponding to the smallest value observed.

dataset→get_summaryStartTime()**YDataSet****dataset→summaryStartTime()**

Returns the start time of the first measure in the data set, relative to the Jan 1, 1970 UTC (Unix timestamp).

When the recording rate is higher than 1 sample per second, the timestamp may have a fractional part.

Returns :

a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

dataset→get_unit()**YDataSet****dataset→unit()**

Returns the measuring unit for the measured value.

js	function get_unit ()
cpp	string get_unit ()
m	-(NSString*) unit
pas	string get_unit (): string
vb	function get_unit () As String
cs	string get_unit ()
java	String get_unit ()
uwp	async Task<string> get_unit ()
py	get_unit ()
php	function get_unit ()
ts	async get_unit (): Promise<string>
es	async get_unit ()
dnp	string get_unit ()
cp	string get_unit ()

Returns :

a string that represents a physical unit.

On failure, throws an exception or returns YDataSet .UNIT_INVALID.

dataset→loadMore()**YDataSet**

Loads the the next block of measures from the dataLogger, and updates the progress indicator.

js	function loadMore ()
cpp	int loadMore ()
m	-(int) loadMore
pas	LongInt loadMore (): LongInt
vb	function loadMore () As Integer
cs	int loadMore ()
java	int loadMore ()
uwp	async Task<int> loadMore ()
py	loadMore ()
php	function loadMore ()
ts	async loadMore (): Promise<number>
es	async loadMore ()
dnp	int loadMore ()
cp	int loadMore ()

Returns :

an integer in the range 0 to 100 (percentage of completion), or a negative error code in case of failure.

On failure, throws an exception or returns a negative error code.

dataset→loadMore_async()**YDataSet**

Loads the the next block of measures from the dataLogger asynchronously.

```
js function loadMore_async( callback, context)
```

Parameters :

- callback** callback function that is invoked when the w The callback function receives three arguments: - the user-specific context object - the YDataSet object whose loadMore_async was invoked - the load result: either the progress indicator (0...100), or a negative error code in case of failure.
- context** user-specific object that is passed as-is to the callback function

Returns :

nothing.

24.8. Class YMeasure

Measured value, returned in particular by the methods of the `YDataSet` class.

`YMeasure` objects are used within the API to represent a value measured at a specified time. These objects are used in particular in conjunction with the `YDataSet` class, but also for sensors periodic timed reports (see `sensor.registerTimedReportCallback`).

In order to use the functions described here, you should include:

js	<code><script type='text/javascript' src='yocto_module.js'></script></code>
cpp	<code>#include "yocto_module.h"</code>
m	<code>#import "yocto_module.h"</code>
pas	<code>uses yocto_module;</code>
vb	<code>yocto_module.vb</code>
cs	<code>yocto_module.cs</code>
java	<code>import com.yoctopuce.YoctoAPI.YMeasure;</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YMeasure;</code>
py	<code>from yocto_module import *</code>
php	<code>require_once('yocto_module.php');</code>
ts	in HTML: <code>import { YMeasure } from '../dist/esm/yocto_module.js';</code> in Node.js: <code>import { YMeasure } from 'yoctolib-cjs/yocto_module.js';</code>
es	in HTML: <code><script src='../lib/yocto_module.js'></script></code> in node.js: <code>require('yoctolib-es2017/yocto_module.js');</code>

YMeasure methods

`measure→get_averageValue()`

Returns the average value observed during the time interval covered by this measure.

`measure→get_endTimeUTC()`

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

`measure→get_maxValue()`

Returns the largest value observed during the time interval covered by this measure.

`measure→get_minValue()`

Returns the smallest value observed during the time interval covered by this measure.

`measure→get_startTimeUTC()`

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

measure→**get_averageValue()****YMeasure****measure**→**averageValue()**

Returns the average value observed during the time interval covered by this measure.

js	function get_averageValue ()
cpp	double get_averageValue ()
m	-(double) averageValue
pas	double get_averageValue (): double
vb	function get_averageValue () As Double
cs	double get_averageValue ()
java	double get_averageValue ()
uwp	double get_averageValue ()
py	get_averageValue ()
php	function get_averageValue ()
ts	get_averageValue (): number
es	get_averageValue ()

Returns :

a floating-point number corresponding to the average value observed.

measure→**get_endTimeUTC()****YMeasure****measure**→**endTimeUTC()**

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

js	function get_endTimeUTC ()
cpp	double get_endTimeUTC ()
m	-(double) endTimeUTC
pas	double get_endTimeUTC (): double
vb	function get_endTimeUTC () As Double
cs	double get_endTimeUTC ()
java	double get_endTimeUTC ()
uwp	double get_endTimeUTC ()
py	get_endTimeUTC ()
php	function get_endTimeUTC ()
ts	get_endTimeUTC (): number
es	get_endTimeUTC ()

When the recording rate is higher than 1 sample per second, the timestamp may have a fractional part.

Returns :

a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the end of this measure.

measure→**get_maxValue()****YMeasure****measure**→**maxValue()**

Returns the largest value observed during the time interval covered by this measure.

js	function get_maxValue ()
cpp	double get_maxValue ()
m	-(double) maxValue
pas	double get_maxValue (): double
vb	function get_maxValue () As Double
cs	double get_maxValue ()
java	double get_maxValue ()
uwp	double get_maxValue ()
py	get_maxValue ()
php	function get_maxValue ()
ts	get_maxValue (): number
es	get_maxValue ()

Returns :

a floating-point number corresponding to the largest value observed.

measure→**get_minValue()****YMeasure****measure**→**minValue()**

Returns the smallest value observed during the time interval covered by this measure.

js	function get_minValue ()
cpp	double get_minValue ()
m	-(double) minValue
pas	double get_minValue (): double
vb	function get_minValue () As Double
cs	double get_minValue ()
java	double get_minValue ()
uwp	double get_minValue ()
py	get_minValue ()
php	function get_minValue ()
ts	get_minValue (): number
es	get_minValue ()

Returns :

a floating-point number corresponding to the smallest value observed.

measure→get_startTimeUTC()**YMeasure****measure→startTimeUTC()**

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

js	function get_startTimeUTC ()
cpp	double get_startTimeUTC ()
m	-(double) startTimeUTC
pas	double get_startTimeUTC (): double
vb	function get_startTimeUTC () As Double
cs	double get_startTimeUTC ()
java	double get_startTimeUTC ()
uwp	double get_startTimeUTC ()
py	get_startTimeUTC ()
php	function get_startTimeUTC ()
ts	get_startTimeUTC (): number
es	get_startTimeUTC ()

When the recording rate is higher than 1 sample per second, the timestamp may have a fractional part.

Returns :

a floating point number corresponding to the number of seconds between the Jan 1, 1970 UTC and the beginning of this measure.

25. Troubleshooting

25.1. Where to start?

If it is the first time that you use a Yoctopuce module and you do not really know where to start, have a look at the Yoctopuce blog. There is a section dedicated to beginners ¹.

25.2. Programming examples don't seem to work

Most of Yoctopuce API programming examples are command line programs and require some parameters to work properly. You have to start them from your operating system command prompt, or configure your IDE to run them with the proper parameters. ².

25.3. Linux and USB

To work correctly under Linux, the the library needs to have write access to all the Yoctopuce USB peripherals. However, by default under Linux, USB privileges of the non-root users are limited to read access. To avoid having to run the *VirtualHub* as root, you need to create a new *udev* rule to authorize one or several users to have write access to the Yoctopuce peripherals.

To add a new *udev* rule to your installation, you must add a file with a name following the "`##-arbitraryName.rules`" format, in the `/etc/udev/rules.d` directory. When the system is starting, *udev* reads all the files with a `".rules"` extension in this directory, respecting the alphabetical order (for example, the `"51-custom.rules"` file is interpreted AFTER the `"50-udev-default.rules"` file).

The `"50-udev-default"` file contains the system default *udev* rules. To modify the default behavior, you therefore need to create a file with a name that starts with a number larger than 50, that will override the system default rules. Note that to add a rule, you need a root access on the system.

In the `udev_conf` directory of the *VirtualHub* for Linux³ archive, there are two rule examples which you can use as a basis.

¹ see: http://www.yoctopuce.com/EN/blog_by_categories/for-the-beginners

² see: <http://www.yoctopuce.com/EN/article/about-programming-examples>

³ <http://www.yoctopuce.com/FR/virtualhub.php>

Example 1: 51-yoctopuce.rules

This rule provides all the users with read and write access to the Yoctopuce USB peripherals. Access rights for all other peripherals are not modified. If this scenario suits you, you only need to copy the "51-yoctopuce_all.rules" file into the "/etc/udev/rules.d" directory and to restart your system.

```
# udev rules to allow write access to all users
# for Yoctopuce USB devices
SUBSYSTEM=="usb", ATTR{idVendor}=="24e0", MODE="0666"
```

Example 2: 51-yoctopuce_group.rules

This rule authorizes the "yoctogroup" group to have read and write access to Yoctopuce USB peripherals. Access rights for all other peripherals are not modified. If this scenario suits you, you only need to copy the "51-yoctopuce_group.rules" file into the "/etc/udev/rules.d" directory and restart your system.

```
# udev rules to allow write access to all users of "yoctogroup"
# for Yoctopuce USB devices
SUBSYSTEM=="usb", ATTR{idVendor}=="24e0", MODE="0664", GROUP="yoctogroup"
```

25.4. ARM Platforms: HF and EL

There are two main flavors of executable on ARM: HF (Hard Float) binaries, and EL (EABI Little Endian) binaries. These two families are not compatible at all. The compatibility of a given ARM platform with one of these two families depends on the hardware and on the OS build. ArmHL and ArmEL compatibility problems are quite difficult to detect. Most of the time, the OS itself is unable to make a difference between an HF and an EL executable and will return meaningless messages when you try to use the wrong type of binary.

All pre-compiled Yoctopuce binaries are provided in both formats, as two separate ArmHF et ArmEL executables. If you do not know what family your ARM platform belongs to, just try one executable from each family.

25.5. Powered module but invisible for the OS

If your Yocto-Watt is connected by USB, if its blue led is on, but if the operating system cannot see the module, check that you are using a true USB cable with data wires, and not a charging cable. Charging cables have only power wires.

25.6. Another process named xxx is already using yAPI

If when initializing the Yoctopuce API, you obtain the *"Another process named xxx is already using yAPI"* error message, it means that another application is already using Yoctopuce USB modules. On a single machine only one process can access Yoctopuce modules by USB at a time. You can easily work around this limitation by using a VirtualHub and the network mode ⁴.

25.7. Disconnections, erratic behavior

If your Yocto-Watt behaves erratically and/or disconnects itself from the USB bus without apparent reason, check that it is correctly powered. Avoid cables with a length above 2 meters. If needed, insert a powered USB hub ^{5 6}.

⁴ see: <http://www.yoctopuce.com/EN/article/error-message-another-process-is-already-using-yapi>

⁵ see: <http://www.yoctopuce.com/EN/article/usb-cables-size-matters>

⁶ see: <http://www.yoctopuce.com/EN/article/how-many-usb-devices-can-you-connect>

25.8. Registering a VirtualHub disconnect an other one

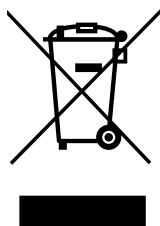
If, when performing a call to `RegisterHub()` with an VirtualHub address, an other previously registered VirtualHub disconnects, make sure the machine running theses VirtualHubs don't have the same *Hostname*. Same *Hostname* can happen very easily when the operating system is installed from a monolithic image, Raspberry-PI are the best example. The Yoctopuce API uses serial numbers to communicate with devices and VirtualHub serial number are created on the fly based the *hostname* of the machine running the VirtualHub.

25.9. Dropped commands

If, after sending a bunch of commands to a Yoctopuce device, you are under the impression that the last ones have been ignored, typical example is a quick and dirty program meant to configure a device, make sure you used a `YAPI.FreeAPI()` at the end of the program. Commands are sent to Yoctopuce modules asynchronously thanks to a background thread. When the main program terminates, that thread is killed no matter if some command are left to be sent. However `API.FreeAPI()` will wait until there is no more commands to send before freeing the API resources and returning.

25.10. Damaged device

Yoctopuce strives to reduce the production of electronic waste. If you believe that your Yocto-Watt 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-Watt, do not throw it away in a trash bin but bring it to your local WEEE recycling point. In this way, it will be disposed properly by a specialized WEEE recycling center.

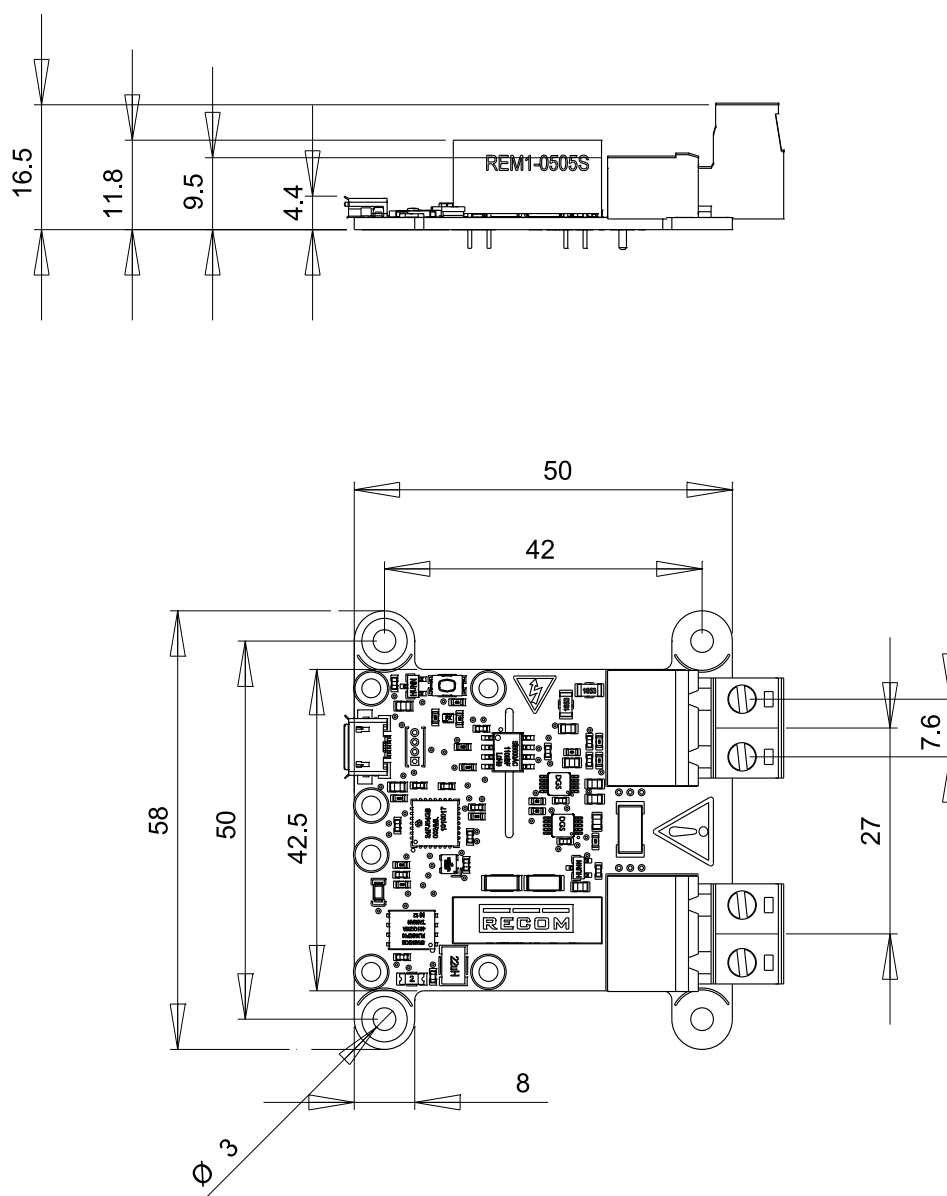
26. Characteristics

You can find below a summary of the main technical characteristics of your Yocto-Watt module.

Product ID	YWATTMK1
Hardware release [†]	Rev. D
USB connector	micro-B
Width	50 mm
Length	58 mm
Weight	22 g
Max Current (continuous)	16 A
Refresh rate	10 Hz
Measuring range (I)	-16...16 A DC/rms
Measuring range (U)	-250...250 V DC/rms
Max working voltage	250 V r.m.s.
Accuracy (AC, 50Hz/60Hz)	0.02 W / 1.5%
Accuracy (DC)	0.002 W / 1.5%
Measurement category	Cat II
Protection class, according to IEC 61140	class II
USB isolation, clearance dist.	3.8 mm
USB isolation, creepage dist.	3.8 / 6 mm
USB isolation, dielectric strength (1 min.)	3 kV r.m.s.
Normal operating temperature	5...40 °C
Extended operating temperature [‡]	-30...85 °C
Power derating	0,67A/°C above 61°C
RoHS compliance	RoHS III (2011/65/UE+2015/863)
USB Vendor ID	0x24E0
USB Device ID	0x002A
Suggested enclosure	YoctoBox-Watt-Transp
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-Watt (rev D)