Yocto-LatchingRelay-C

User's guide

Table of contents

1. Introduction

The Yocto-LatchingRelay-C module is a small 54x20mm module which allows you to control a small relay by USB. This relay can commute up to 250V and 5A, which allows you to pilot numerous equipments by acting directly on their power supply. The module small dimensions enable it to be slipped almost anywhere, including inside the piloted equipment.

The Yocto-LatchingRelay-C module contains a latching relay: this relay keeps its state even when the power is cut off. Moreover the Yocto-LatchingRelay-C features a button and pads allowing you to switch the relay manually
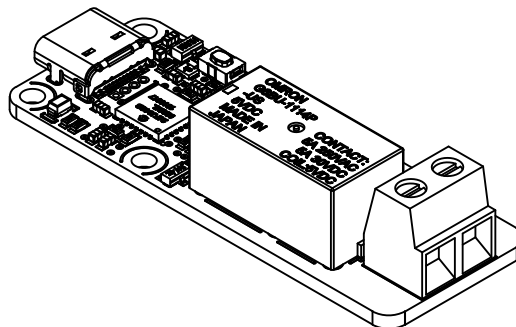
The Yocto-LatchingRelay-C module

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

Yoctopuce thanks you for buying this Yocto-LatchingRelay-C and sincerely hopes that you will be satisfied with it. The Yoctopuce engineers have put a large amount of effort to ensure that your Yocto-LatchingRelay-C is easy to install anywhere and easy to drive from a maximum of programming languages. If you are nevertheless disappointed with this module, or if you need additional information, do not hesitate to contact Yoctopuce support:

E-mail address:	support@yoctopuce.com
Web site:	www.yoctopuce.com
Postal address:	Route de Cartigny 33
ZIP code, city:	1236 Cartigny
Country:	Switzerland

1.1. Safety Information

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

Protective enclosure

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

Maintenance

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

Identification

In order to ease the maintenance and the identification of risks during maintenance, you should stick the water-resistant identification label provided together with the electronic board as close as possible to the device. If the device is put in a dedicated enclosure, the identification label should be affixed on the outside of the enclosure. This label is resistant to humidity and to the usual rubbing that can occur during normal maintenance.
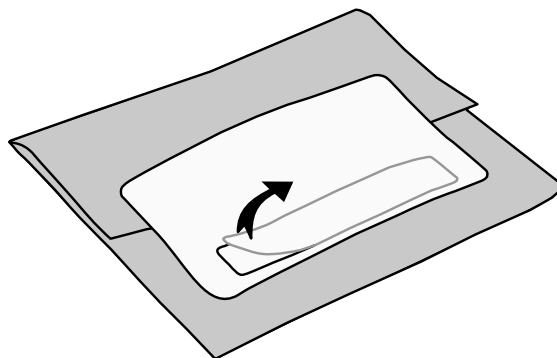
Identification label is integrated in the package label.

Application

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

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

Environment

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

IEC 61140 Protection Class II
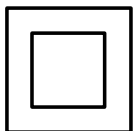
This equipment is protected throughout by a double insulation or a reinforced insulation. The electronics on the side of USB connector are designed for safety extra-low voltages, while the electronics on the opposite side may carry hazardous voltages.

Electrical shock
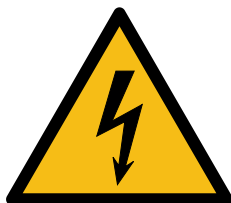
Voltages above 60 V DC, 30 V AC r.m.s. or 42.4 V peak create electrocution and fire hazards. Since the Yocto-LatchingRelay-C has bare contacts, you should not connect it to hazardous voltage without using a proper enclosure, electrically insulated, non-flammable and shock-proof as described previously. 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-LatchingRelay-C 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-LatchingRelay-C 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-LatchingRelay-C 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-LatchingRelay-C 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-LatchingRelay-C 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-LatchingRelay-C 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-LatchingRelay-C 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-LatchingRelay-C 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-LatchingRelay-C is -25..70°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-LatchingRelay-C in harsh environments for a long period of time, we strongly advise you to run extensive tests before going into production.

2. Presentation

1: USB socket   5: Manual switching pads
2: Yocto-button   6: Led of output BOutput B
3: Yocto-led   7: Relay input/output
4: Manual switching button
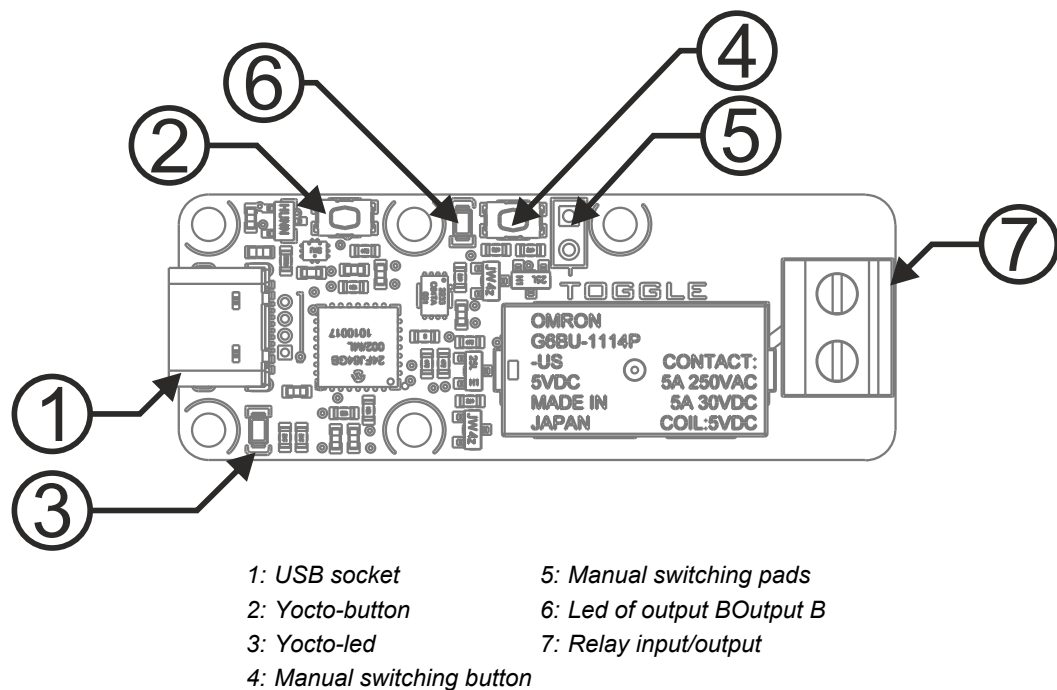
2.1. Common elements

All Yocto-modules share a number of common functionalities.

USB connector

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

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

- **Voltage min.:** 4.75 V DC

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

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

Yocto-button

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

Yocto-led

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

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

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

Current sensor

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

Serial number

Each Yocto-module has a unique serial number assigned to it at the factory. For Yocto-LatchingRelay-C modules, this number starts with LTCHRL2C. The module can be software driven using this serial number. The serial number cannot be modified.

Logical name

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

2.2. Specific elements

Screw terminal

The relay embedded in the Yocto-LatchingRelay-C works as a simple normally-open switch. It means that when they are at rest (state A), the commanded circuit is open; when the relay is active (state B), the circuit is closed.

---

[1] short-short-short long-long-long short-short-short
[2] support@yoctopuce.com
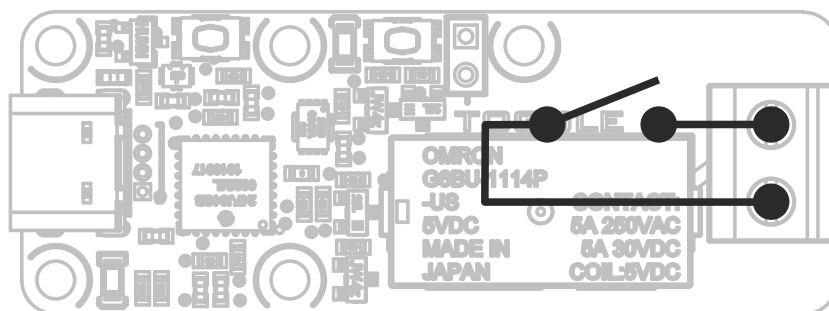
Relay wiring inside the module.

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 26 AWG and 16 AWG, meaning 0.13 mm^2 to 1.5 mm^2. The wire must be stripped on 5mm, and the recommended tightening torque is 0.4 Nm.

Relay

The relay itself is a miniature electromechanical power relays G6BU-1114P with high dielectric strength, made by OMRON. Be aware that like for all electromechanical relays, when used with direct current (DC), the switching capacity goes progressivly down while the voltage goes up:

Maximum Switching Capacity
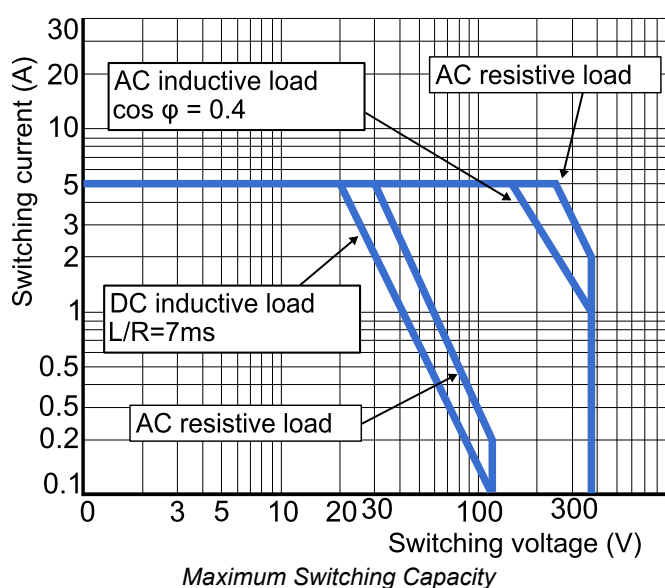
The contacts durability depends from the current and voltage as well, and also from the type of load:

Relay Durability
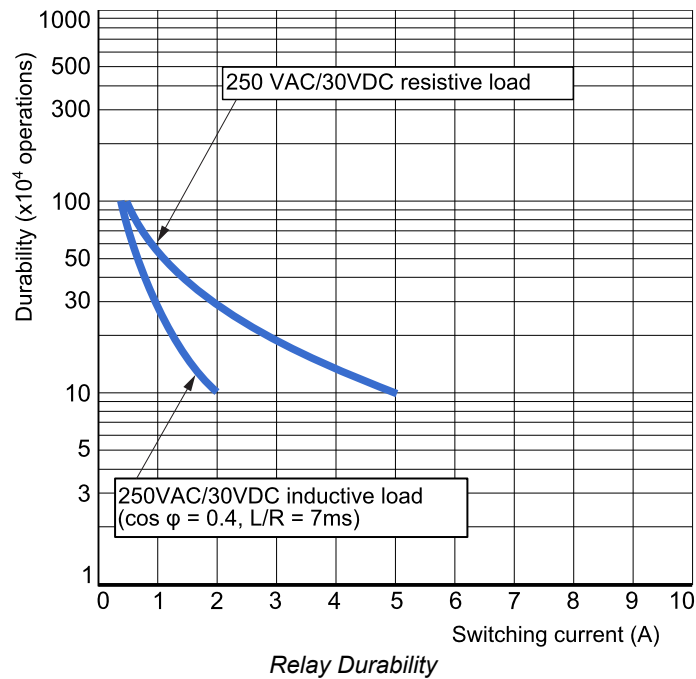
The relays are not impacted by vertical and lateral shocks up to 980 m/s^2, but shocks colinear with the relay length may have more impact:

Shock Malfunction
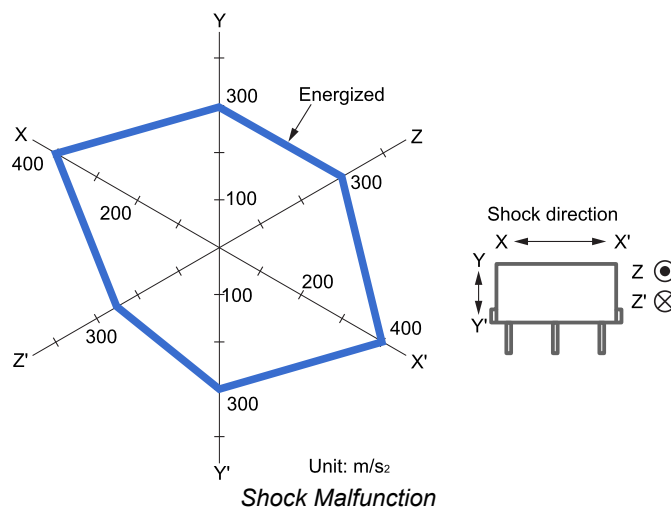
Led indicating the active output

On the device, there is a green leds which indicates that the output is active. By default, the light of this led is rather strong, but you can change its luminosity by software.

2.3. Safety isolation

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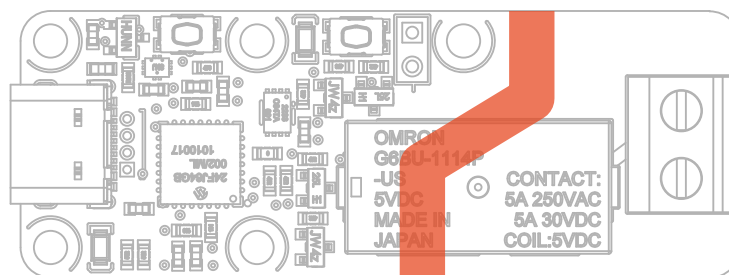
Yocto-LatchingRelay-C safety isolation

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-LatchingRelay-C 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-LatchingRelay-C, 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.

The specifications of the safety isolation are as follows:

- **Test voltage:** 3kV
- **Clearance distance:** 4.1mm
- **Creepage distance:** 5mm
- **Material group:** Cat IIIa

The specifications of the isolation between output contacts are as follow:

- **Dielectric strength:** 1kV (1 min)
- **Clearance distance:** 2.9mm
- **Creepage distance:** 3.8mm
- **Material group:** Cat IIIa

2.4. Optional accessories

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

Screws and spacers

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

Micro-USB hub

If you intend to put several Yoctopuce modules in a very small space, you can connect them directly to a micro-USB hub. Yoctopuce builds a multi-TT USB hub particularly small for this purpose (down

to 20mmx36mm), on which you can directly solder a USB cable instead of using a USB plug. For more details, see the micro-USB hub information sheet.

YoctoHub-Ethernet, YoctoHub-Wireless and YoctoHub-GSM

You can add network connectivity to your Yocto-LatchingRelay-C, thanks to the YoctoHub-Ethernet, the YoctoHub-Wireless and the YoctoHub-GSM which provides respectively Ethernet, WiFi and GSM connectivity. All of them can drive up to three devices and behave exactly like a regular computer running the VirtualHub application[3].

1.27mm (or 1.25mm) connectors

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

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

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

Enclosure

Your Yocto-LatchingRelay-C has been designed to be installed as is in your project. Nevertheless, Yoctopuce sells enclosures specifically designed for Yoctopuce devices. These enclosures have removable mounting brackets and magnets allowing them to stick on ferromagnetic surfaces. More details are available on the Yoctopuce web site[4]. The suggested enclosure model for your Yocto-LatchingRelay-C is the Yoctobox-long-thick-black.
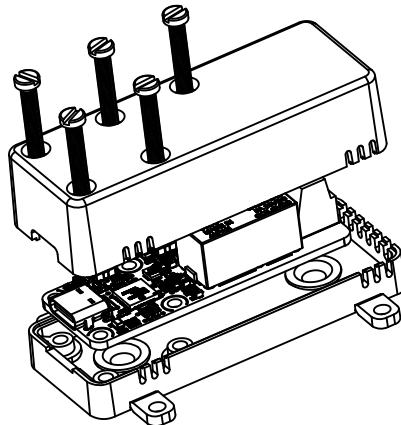
You can install your Yocto-LatchingRelay-C in an optional enclosure

---

[3] http://www.yoctopuce.com/EN/virtualhub.php
[4] http://www.yoctopuce.com/EN/products/category/enclosures

3. First steps

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

3.1. Prerequisites

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

A computer

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

Yoctopuce provides software libraries to drive its modules for the following operating systems: **Windows, Linux, macOS, and Android**. Yoctopuce modules do not require the installation of specific drivers, as they use the HID driver[1] standardly supplied in all operating systems.

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

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

The architectures supported by Yoctopuce software libraries are as follows:

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

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

---

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

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

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

An USB cable, type A - USB-C

USB connectors come in several shapes. The "standard" size is the one you probably use to connect your printer. The "mini" size has more or less disappeared. The "micro" size was the smallest when the first Yoctopuce modules were designed. Over the last few years, USB-C connectors have appeared and are about replace all other types. That is why, since 2024 Yoctopuce, is progressively migrating its product line to USB-C[2].
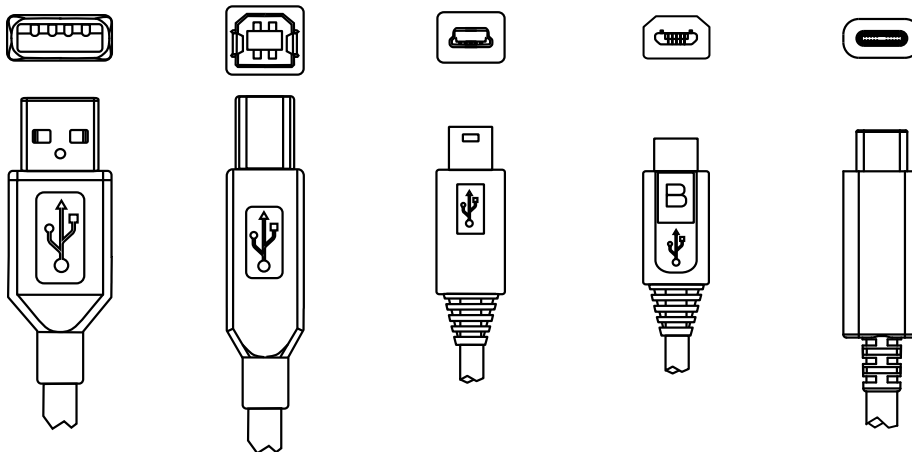
The most common USB 2.0 connectors: A, B, Mini B, Micro B and USB-C.

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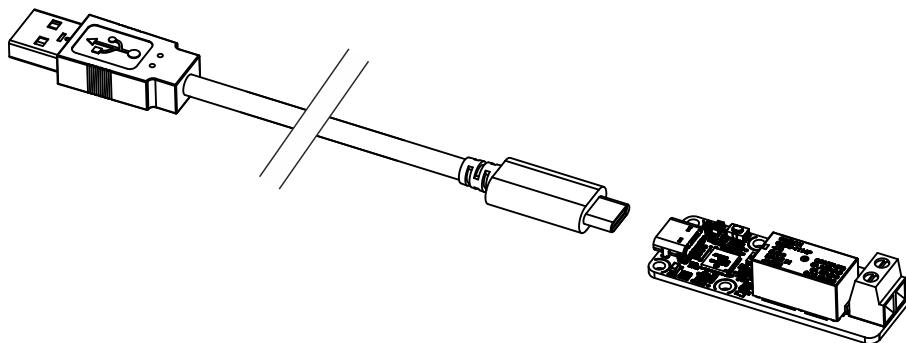
You must plug in your Yocto-LatchingRelay-C module with a USB 2.0 cable of type A-USB-C

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

3.2. Testing USB connectivity

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

---

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

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

Module list as displayed in your web bowser

3.3. Localization

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

3.4. Test of the module

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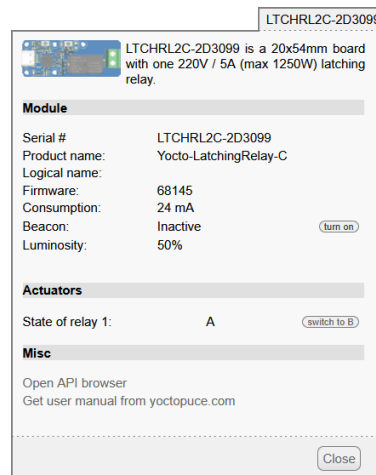
Properties of the Yocto-LatchingRelay-C module

This window allows you, among other things, to test the relay with the **switch to A** / **switch to B** button. There is a characteristic clicking when the relay is working. Moreover, the led indicating the active output lights up. Notice that manual state changes caused by the button on the board are detected.

3.5. Configuration

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

---

[3] www.yoctopuce.com/EN/virtualhub.php
[4] The interface is tested on Chrome, FireFox, Safari, Edge et IE 11.
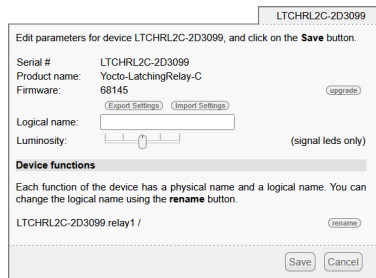
Yocto-LatchingRelay-C module configuration

Firmware

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

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

Logical name of the module

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

Luminosity

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

Logical names of functions

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

The only function of the Yocto-LatchingRelay-C module is its embedded relay, with the hardware name "relay1".

---

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

4. Assembly and connections

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

4.1. Fixing

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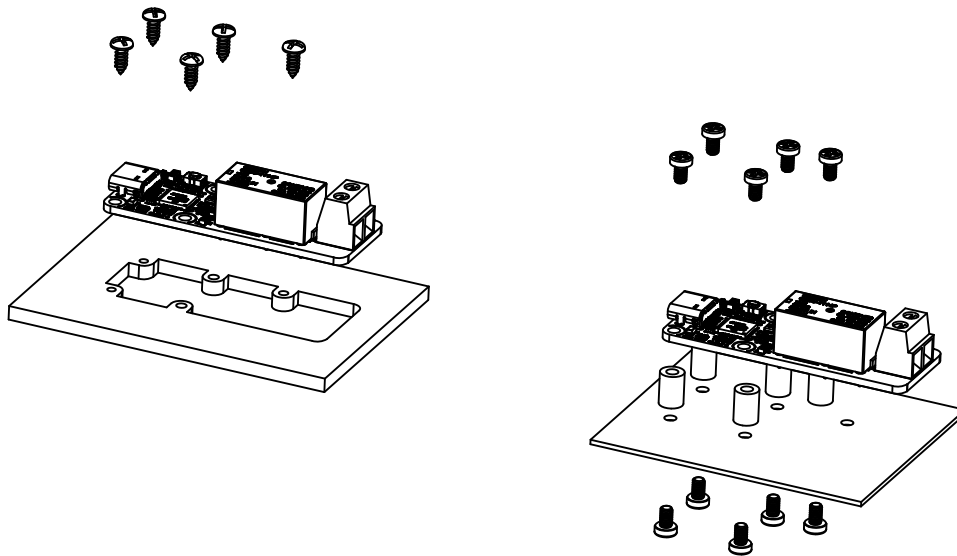
Examples of assembly on supports

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

4.2. Assembly examples

If you obtained this Yocto-LatchingRelay-C module, it is probably because you know exactly what you intend to do with it. You can nevertheless find below a wiring examples, among the simplest.
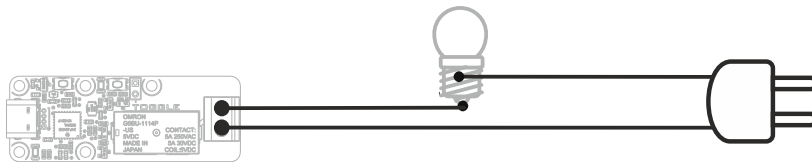
Pilot a light bulb with your Yocto-LatchingRelay-C module.

4.3. Manual switching

The Yocto-LatchingRelay-C can be switched manually.The relay switches when the the matching button is pressed or when the contacts near this buttons are briefly shorted. So you just have to wire as many switches as necessary to drive the Yocto-LatchingRelay-C both manually and with a computer.
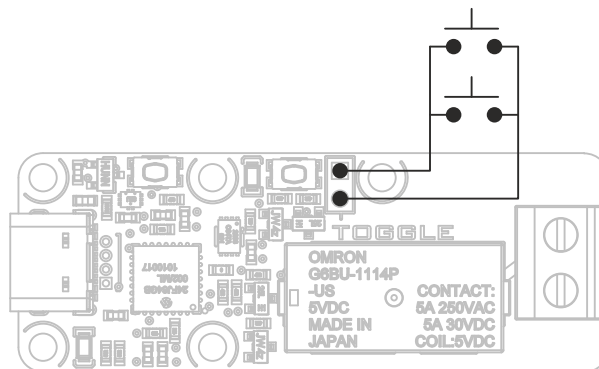
Switch wiring for manual control of the Yocto-LatchingRelay-C .

4.4. Electro-magnetic relays and coils

Some devices that you may wish to control with your Yocto-LatchingRelay-C module contain large coils. It is in particular the case for electric motors and transformers. This may cause trouble because of the auto-induction generated when current goes through a coil. A very high voltage briefly appears at the ends of a coil when when one brutally cuts the current passing through it. This high voltage can create an electric arc where the circuit was cut, in our case inside the relay soldered on the module. This electric arc can eat away the relay connections, leading to premature aging.

Therefore, we advise against controlling electric motors or transformers with an electro-magnetic relay, be it a Yocto-LatchingRelay-C module, or any other command system based on this technology.

It is possible to limit this phenomenon by inserting a TVS diode in parallel with the device containing the coil. These diodes are blocking below a specified voltage, and conductive beyond. So if you clamp one of theses diode on your inductive load, this will short cut voltage peaks. You only have to choose a diode with the right clamping voltage for your application. If you wish to know more, Tyco published an application note about this [1].
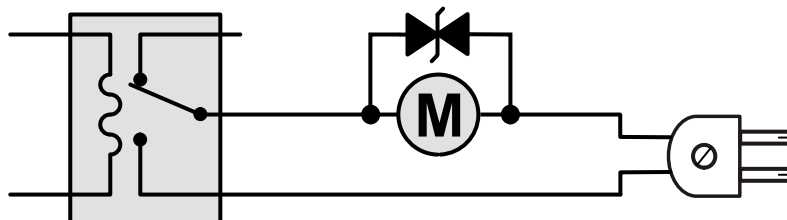
Controlling an electric motor with a relay, using a protecting diode.

---

[1] Relay contact life, Application note, Tyco electronics, http://relays.te.com/appnotes/app_pdfs/13c3236.pdf

4.5. 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 [2]. Good cables are usually made using AWG 26 or AWG 28 wires for data lines and AWG 24 wires for power.

4.6. Electromagnetic compatibility (EMI)

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

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

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

---

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

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

5. Programming, general concepts

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

5.1. Programming paradigm

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

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

Low level handling   Module handling   Feature handling

YAPI		YModule		YTemperature
				YPressure
				YRelay
				YXxx

Structure of the Yoctopuce API
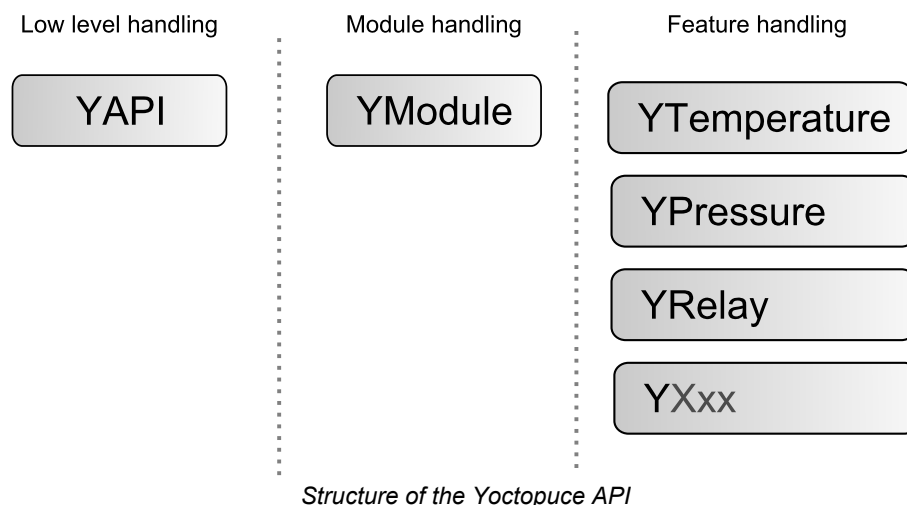
The YSensor class

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

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

Programmation

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

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

Access to the functions of a module

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

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

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

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

Function handling

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

Access to the modules

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

Functions/Module interaction

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

5.2. The Yocto-LatchingRelay-C module

The Yocto-LatchingRelay-C module provides a single instances of the Relay function, corresponding to the only relay present on the module.

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

relay1 : Relay

attribute	type	modifiable ?
logicalName	String	modifiable
advertisedValue	String	modifiable
state	A/B	modifiable
stateAtPowerOn	Enumerated	modifiable
maxTimeOnStateA	Time	modifiable
maxTimeOnStateB	Time	modifiable
output	On/Off	modifiable
pulseTimer	Time	modifiable
delayedPulseTimer	Aggregate	modifiable
countdown	Time	read-only

5.3. 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-LatchingRelay-C module, this serial number always starts with LTCHRL2C. You can use the serial number to access a given module by software.

logicalName

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

productId

USB device identifier of the module, preprogrammed to 217 at the factory.

productRelease

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

firmwareRelease

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

persistentSettings

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

luminosity

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

beacon

Activity of the localization beacon of the module.

upTime

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

usbCurrent

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

rebootCountdown

Countdown to use for triggering a reboot of the module.

userVar

32bit integer variable available for user storage.

5.4. Relay

relay control interface, available for instance in the Yocto-LatchedRelay, the Yocto-MaxiPowerRelay, the Yocto-PowerRelay-V3 or the Yocto-Relay

The `YRelay` class allows you to drive a Yoctopuce relay or optocoupled output. It can be used to simply switch the output on or off, but also to automatically generate short pulses of determined duration. On devices with two output for each relay (double throw), the two outputs are named A and B, with output A corresponding to the idle position (normally closed) and the output B corresponding to the active state (normally open).

logicalName

Character string containing the logical name of the relay, initially empty. This attribute can be modified at will by the user. Once initialized to an non-empty value, it can be used to access the relay directly. If two relays 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 relay, that will be automatically advertised up to the parent hub. For a relay, the advertised value is the the relays state (A for the idle position, B for the active position).

state

Active output of the relays: A for the idle position, B for the active position.

stateAtPowerOn

Active output of the relays at device power on: A for the idle position, B for the active position, UNCHANGED to leave the relay as is.

maxTimeOnStateA

Maximum time (ms) allowed for relays to stay in state A before automatically switching back in to B state. Zéro means no maximum time.

maxTimeOnStateB

Maximum time (ms) allowed for relays to stay in state B before automatically switching back in to A state. Zéro means no maximum time.

output

Output state of the relays, when used as a simple switch (single throw).

pulseTimer

Time during which the relays should be kept in state B (active) before returning automatically to state A (idle state). Any explicit state change issued afterwards will cancel the automated switch.

delayedPulseTimer

Delayed pulse parameters.

countdown

Waiting delay before next pulse (delayed pulse case).

5.5. What interface: Native, DLL or Service ?

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

Native control

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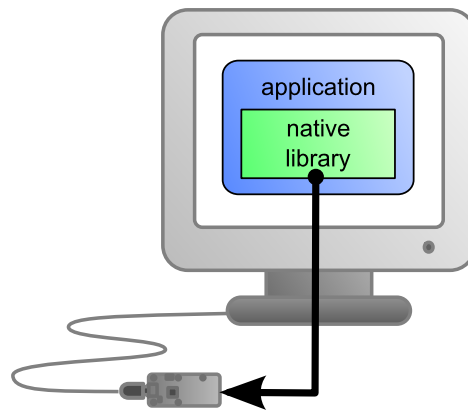
The application uses the native library to control the locally connected module

Native control by DLL

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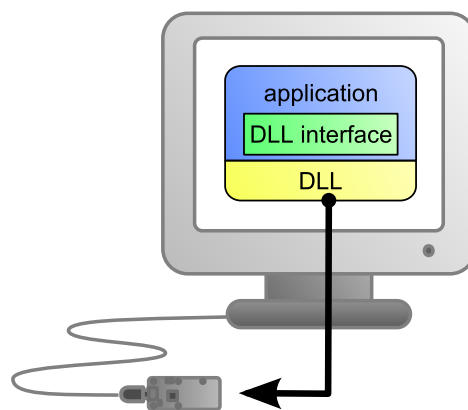
The application uses the DLL to natively control the locally connected module

Control by service

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

---

[1] www.yoctopuce.com/EN/virtualhub.php
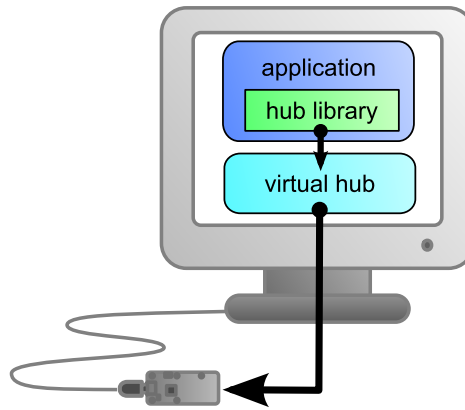
The application connects itself to VirtualHub to gain access to the module

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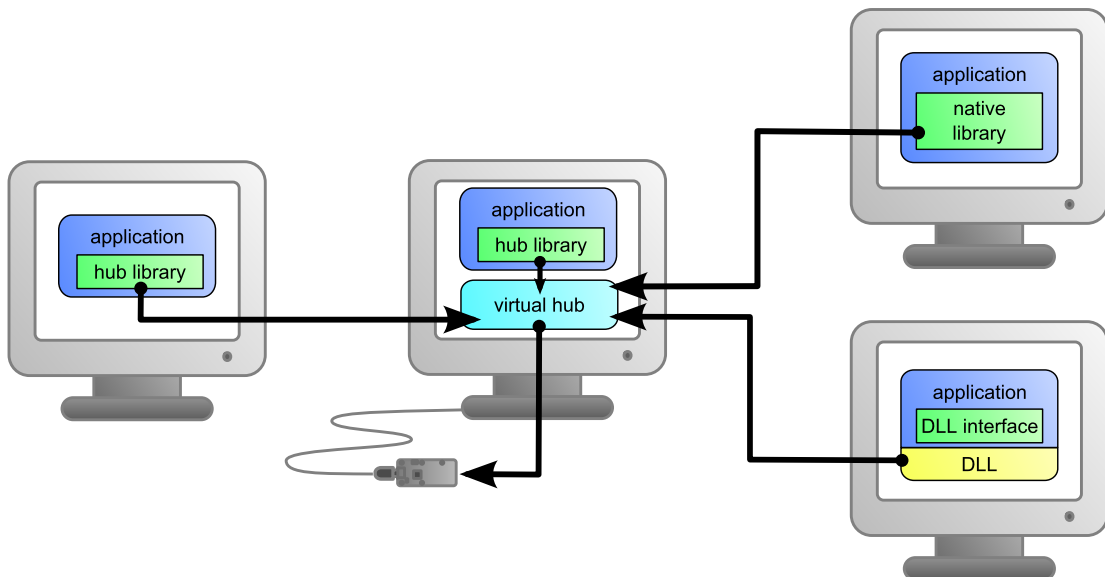
When VirtualHub is used, the control application does not need to reside on the same machine as the module.

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

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

Support methods for different languages

5.6. Accessing modules through a hub

VirtualHub to work around USB access limitation

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

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

```
Another process is already using yAPI
```

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

With a YoctoHub

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

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

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

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

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

5.7. Programming, where to start?

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

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

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

---

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

6. Using the Yocto-LatchingRelay-C in command line

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

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

6.1. Installing

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

```
C:\>YRelay any set_state B
```

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

6.2. Use: general description

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

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

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

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

---

[1] If you want to recompile the command line API, you also need the C++ API.
[2] http://www.yoctopuce.com/EN/libraries.php

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

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

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

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

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

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

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

6.3. Control of the Relay function

To control the Relay function of your Yocto-LatchingRelay-C, you need the YRelay executable file.

For instance, you can launch:

```
C:\>YRelay any set_state B
```

This example uses the "*any*" target to indicate that we want to work on the first Relay 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-LatchingRelay-C module with the *LTCHRL2C-123456* serial number which you have called "*MyModule*", and its relay1 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:\>YRelay LTCHRL2C-123456.relay1 describe

C:\>YRelay LTCHRL2C-123456.MyFunction describe

C:\>YRelay MyModule.relay1 describe

C:\>YRelay MyModule.MyFunction describe

C:\>YRelay MyFunction describe
```

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

```
C:\>YRelay all describe
```

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

```
C:\>YRelay /help
```

6.4. Control of the module part

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

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

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

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

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

```
C:\>YModule LTCHRL2C-12346 set_logicalName MonPremierModule

C:\>YModule LTCHRL2C-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 LTCHRL2C-12346 set_logicalName MonPremierModule
C:\>YModule LTCHRL2C-12346 saveToFlash
```

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

```
C:\>YModule -s  LTCHRL2C-12346 set_logicalName MonPremierModule
```

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

6.5. Limitations

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

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

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

you obtain a list of the modules connected by USB, using a native access. If another command which accesses the modules natively is already running, this does not work. But if you run VirtualHub, 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 VirtualHub. Note that VirtualHub counts as a native application.

---

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

7. Using the Yocto-LatchingRelay-C with Python

Python is an interpreted object oriented language developed by Guido van Rossum. Among its advantages is the fact that it is free, and the fact that it is available for most platforms, Windows as well as UNIX. It is an ideal language to write small scripts on a napkin. The Yoctopuce library is compatible with Python 2.7 and 3.x up to the latest official versions. It works under Windows, macOS, and Linux, Intel as well as ARM. Python interpreters are available on the Python web site[1].

7.1. Source files

The Yoctopuce library classes[2] for Python that you will use are provided as source files. Copy all the content of the *Sources* directory in the directory of your choice and add this directory to the *PYTHONPATH* environment variable. If you use an IDE to program in Python, refer to its documentation to configure it so that it automatically finds the API source files.

7.2. Dynamic library

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

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

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

7.3. Control of the Relay function

A few lines of code are enough to use a Yocto-LatchingRelay-C. Here is the skeleton of a Python code snipplet to use the Relay 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
relay = YRelay.FindRelay("LTCHRL2C-123456.relay1")

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

[...]
```

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.

YRelay.FindRelay

The `YRelay.FindRelay` function allows you to find a relay 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-LatchingRelay-C module with serial number *LTCHRL2C-123456* which you have named "*MyModule*", and for which you have given the *relay1* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
relay = YRelay.FindRelay("LTCHRL2C-123456.relay1")
relay = YRelay.FindRelay("LTCHRL2C-123456.MyFunction")
relay = YRelay.FindRelay("MyModule.relay1")
relay = YRelay.FindRelay("MyModule.MyFunction")
relay = YRelay.FindRelay("MyFunction")
```

`YRelay.FindRelay` returns an object which you can then use at will to control the relay.

isOnline

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

About python imports

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

```
from yocto_api import *
```

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

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

set_state

The `set_state()` method of the objet returned by `YRelay.FindRelay` switches the relay position to one of its two outputs. The two possible parameter values are `YRelay.STATE_A` for output A, and `YRelay.STATE_B` for output B.

A real example

Launch Python and open the corresponding sample script provided in the directory **Examples/Doc-GettingStarted-Yocto-LatchingRelay-C** of the Yoctopuce library.

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

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

from yocto_api import *
from yocto_relay import *


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


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


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

target = sys.argv[1].upper()
state = sys.argv[2].upper()

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

if target == 'ANY':
    # retreive any Relay
    relay = YRelay.FirstRelay()
    if relay is None:
        die('no device connected')
else:
    relay = YRelay.FindRelay(target + ".relay1")

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

if state == 'A':
    relay.set_state(YRelay.STATE_A)
else:
    relay.set_state(YRelay.STATE_B)
YAPI.FreeAPI()
```

7.4. Control of the module part

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

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

from yocto_api import *


def usage():
```

```python
    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.

```python
#!/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.

```python
#!/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()
```

7.5. Error handling

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

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

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

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

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

library to a mode where exceptions for all the functions are systematically replaced by specific return values, which can be tested by the caller when necessary. For each function, the name of each return value in case of error is systematically documented in the library reference. The name always follows the same logic: a `get_state()` method returns a `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.

8. Using Yocto-LatchingRelay-C with C++

C++ is not the simplest language to master. However, if you take care to limit yourself to its essential functionalities, this language can very well be used for short programs quickly coded, and it has the advantage of being easily ported from one operating system to another. Under Windows, C++ is supported with Microsoft Visual Studio 2017 and more recent versions. Under macOS, we support the XCode versions supported by Apple. And under Linux, we support all GCC version published since 2008. Moreover, under Max OS X and under Linux, you can compile the examples using a command line with GCC using the provided `GNUmakefile`. In the same manner under Windows, a `Makefile` allows you to compile examples using a command line, fully knowing the compilation and linking arguments.

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

8.1. Control of the Relay function

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

```cpp
#include "yocto_api.h"
#include "yocto_relay.h"

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

// Retrieve the object used to interact with the device
YRelay *relay;
```

---

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

```
relay = YRelay::FindRelay("LTCHRL2C-123456.relay1");

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

Let's look at these lines in more details.

yocto_api.h et yocto_relay.h

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

YAPI::RegisterHub

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

YRelay::FindRelay

The `YRelay::FindRelay` function allows you to find a relay 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-LatchingRelay-C module with serial number *LTCHRL2C-123456* which you have named "*MyModule*", and for which you have given the *relay1* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
YRelay *relay = YRelay::FindRelay("LTCHRL2C-123456.relay1");
YRelay *relay = YRelay::FindRelay("LTCHRL2C-123456.MyFunction");
YRelay *relay = YRelay::FindRelay("MyModule.relay1");
YRelay *relay = YRelay::FindRelay("MyModule.MyFunction");
YRelay *relay = YRelay::FindRelay("MyFunction");
```

`YRelay::FindRelay` returns an object which you can then use at will to control the relay.

isOnline

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

set_state

The `set_state()` method of the objet returned by `yFindRelay` switches the relay position to one of its two outputs. The two possible parameter values are `Y_STATE_A` for output A, and `Y_STATE_B` for output B.

A real example

Launch your C++ environment and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-LatchingRelay-C** of the Yoctopuce library. If you prefer to work with your favorite text editor, open the file `main.cpp`, and type `make` to build the example when you are done.

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

```
#include "yocto_api.h"
#include "yocto_relay.h"
#include <iostream>
#include <ctype.h>
#include <stdlib.h>
```

```cpp
using namespace std;

static void usage(void)
{
  cout << "usage: demo <serial_number>  [ A | B ]" << endl;
  cout << "       demo <logical_name> [ A | B ]" << endl;
  cout << "       demo any [ A | B ]" << 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;
  string  target;
  YRelay  *relay;
  char    state;

  if (argc < 3) {
    usage();
  }
  target = (string) argv[1];
  state = toupper(argv[2][0]);

  // 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") {
    relay = YRelay::FirstRelay();
    if (relay == NULL) {
      cout << "No module connected (check USB cable)" << endl;
      return 1;
    }
  } else {
    relay =  YRelay::FindRelay(target + ".relay1");
  }

  if (relay->isOnline()) {
    relay->set_state(state == 'A' ? Y_STATE_A : Y_STATE_B);
  } else {
    cout << "Module not connected (check identification and USB cable)" << endl;
  }
  YAPI::FreeAPI();

  return 0;
}
```

8.2. 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.

```cpp
#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);
}
```

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

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

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

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

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

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

Changing the module settings

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

```cpp
#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;
}
```

8.3. 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.

8.4. 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 macOS: **IOKit.framework** and **CoreFoundation.framework**
- For Linux: **libm**, **libpthread**, **libusb1.0**, and **libstdc++**

Integration as a static library

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

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

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

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

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

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

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

Integration as a dynamic library

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

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

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

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

- For Windows: **yocto.lib**

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

With GCC, the command line to compile is simply:

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

9. Using Yocto-LatchingRelay-C with C#

C# (pronounced C-Sharp) is an object-oriented programming language promoted by Microsoft, it is somewhat similar to Java. Like Visual-Basic and Delphi, it allows you to create Windows applications quite easily. C# is supported under Windows Visual Studio 2017 and its more recent versions.

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

9.1. Installation

Download the Visual C# Yoctopuce library from the Yoctopuce web site[1]. 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.

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

The Visual C#.NET Yoctopuce library is composed of a DLL and of source files in Visual C#. The DLL is not a .NET DLL, but a classic DLL, written in C, which manages the low level communications with the modules[2]. The source files in Visual C# manage the high level part of the API. Therefore, your need both this DLL and the .cs files of the `sources` directory to create a project managing Yoctopuce modules.

Configuring a Visual C# project

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

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

---

[1] www.yoctopuce.com/EN/libraries.php
[2] The sources of this DLL are available in the C++ API

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

Then add in the same manner the `yapi.dll` DLL, located in the `Sources/dll` directory[3]. 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.

9.3. Control of the Relay function

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

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

// Retrieve the object used to interact with the device
YRelay relay = YRelay.FindRelay("LTCHRL2C-123456.relay1");

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

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.

YRelay.FindRelay

The `YRelay.FindRelay` function allows you to find a relay 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-LatchingRelay-C module with serial number *LTCHRL2C-123456* which you have named "*MyModule*", and for which you have given the *relay1* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
relay = YRelay.FindRelay("LTCHRL2C-123456.relay1");
relay = YRelay.FindRelay("LTCHRL2C-123456.MyFunction");
relay = YRelay.FindRelay("MyModule.relay1");
relay = YRelay.FindRelay("MyModule.MyFunction");
relay = YRelay.FindRelay("MyFunction");
```

`YRelay.FindRelay` returns an object which you can then use at will to control the relay.

---

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

isOnline

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

set_state

The `set_state()` method of the objet returned by `YRelay.FindRelay` switches the relay position to one of its two outputs. The two possible parameter values are `YRelay.STATE_A` for output A, and `YRelay.STATE_B` for output B.

A real example

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

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

```csharp
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_number>  [ A | B ]");
      Console.WriteLine(execname + "  <logical_name>  [ A | B ]");
      Console.WriteLine(execname + "  any [ A | B ]");
      System.Threading.Thread.Sleep(2500);
      Environment.Exit(0);
    }

    static void Main(string[] args)
    {
      string errmsg = "";
      string target;
      YRelay relay;
      string state;

      if (args.Length < 2) usage();
      target = args[0].ToUpper();
      state = args[1].ToUpper();

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

      if (target == "ANY") {
        relay = YRelay.FirstRelay();
        if (relay == null) {
          Console.WriteLine("No module connected (check USB cable) ");
          Environment.Exit(0);
        }
      } else relay = YRelay.FindRelay(target + ".relay1");

      if (relay.isOnline()) {
        if (state == "A") {
          relay.set_state(YRelay.STATE_A);
        } else {
          relay.set_state(YRelay.STATE_B);
        }
      } else {
        Console.WriteLine("Module not connected");
        Console.WriteLine("check identification and USB cable");
      }
      YAPI.FreeAPI();
```

```
      }
    }
}
```

9.4. Control of the module part

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

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


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

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

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


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

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

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

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

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

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

Changing the module settings

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

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

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

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

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

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

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

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

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

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

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

Listing the modules

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

```csharp
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();
    }
  }
}
```

9.5. Error handling

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

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

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

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

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

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

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

10. Using the Yocto-LatchingRelay-C with LabVIEW

LabVIEW is edited by National Instruments since 1986. It is a graphic development environment: rather than writing lines of code, the users draw their programs, somewhat like a flow chart. LabVIEW was designed mostly to interface measuring tools, hence the *Virtual Instruments* name for LabVIEW programs. With visual programming, drawing complex algorithms becomes quickly fastidious. The LabVIEW Yoctopuce library was thus designed to make it as easy to use as possible. In other words, LabVIEW being an environment extremely different from other languages supported by Yoctopuce, there are major differences between the LabVIEW API and the other APIs.

10.1. Architecture

The LabVIEW library is based on the Yoctopuce DotNetProxy library contained in the DotNetProxyLibrary.dll DLL. In fact, it is this DotNetProxy library which takes care or most of the work by relying on the C# library which, in turn, uses the low level library coded in yapi.dll (32bits) and amd64\yapi.dll(64bits).
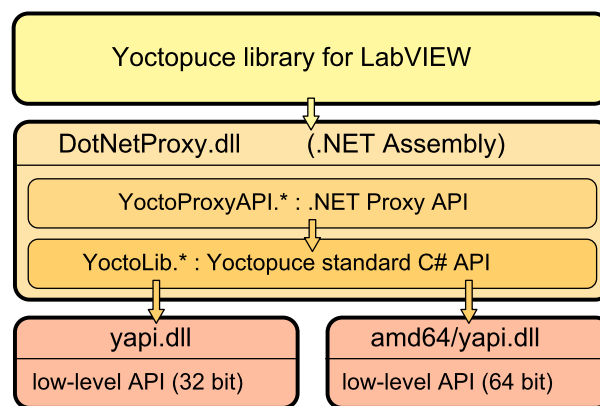
LabVIEW Yoctopuce API architecture

You must therefore imperatively distribute the *DotNetProxyLibrary.dll*, *yapi.dll*, and *amd64\yapi.dll* with your LabVIEW applications using the Yoctopuce API.

If need be, you can find the low level API sources in the C# library and the *DotNetProxyLibrary.dll* sources in the *DotNetProxy* library.

10.2. Compatibility

Firmware

For the LabVIEW Yoctopuce library to work correctly with your Yoctopuce modules, these modules need to have firmware 37120, or higher.

LabVIEW for Linux and MacOS

At the time of writing, the LabVIEW Yoctopuce API has been tested under Windows only. It is therefore most likely that it simply does not work with the Linux and MacOS versions of LabVIEW.

LabVIEW NXG

The LabVIEW Yoctopuce library uses many techniques which are not yet available in the new generation of LabVIEW. The library is therefore absolutely not compatible with LabVIEW NXG.

About DotNewProxyLibrary.dll

In order to be compatible with as many versions of Windows as possible, including Windows XP, the *DotNetProxyLibrary.dll* library is compiled in .NET 3.5, which is available by default on all the Windows versions since XP.

10.3. Installation

Download the LabVIEW library from the Yoctopuce web site[1]. It is a ZIP file in which there is a distinct directory for each version of LabVIEW. Each of these directories contains two subdirectories: the first one contains programming examples for each Yoctopuce product; the second one, called *VIs*, contains all the VIs of the API and the required DLLs.

Depending on Windows configuration and the method used to copy the *DotNetProxyLibrary.dll* on your system, Windows may block it because it comes from an other computer. This may happen when the library zip file is uncompressed with Window's file explorer. If the DLL is blocked, LabVIEW will not be able to load it and an error 1386 will occur whenever any of the Yoctopuce VIs is executed.

There are two ways to fix this. The simplest is to unblock the file with the Windows file explorer: *right click / properties* on the *DotNetProxyLibrary.dll* file, and click on the *unblock* button. But this has to be done each time a new version of the DLL is copied on your system.

---

[1] http://www.yoctopuce.com/EN/libraries.php
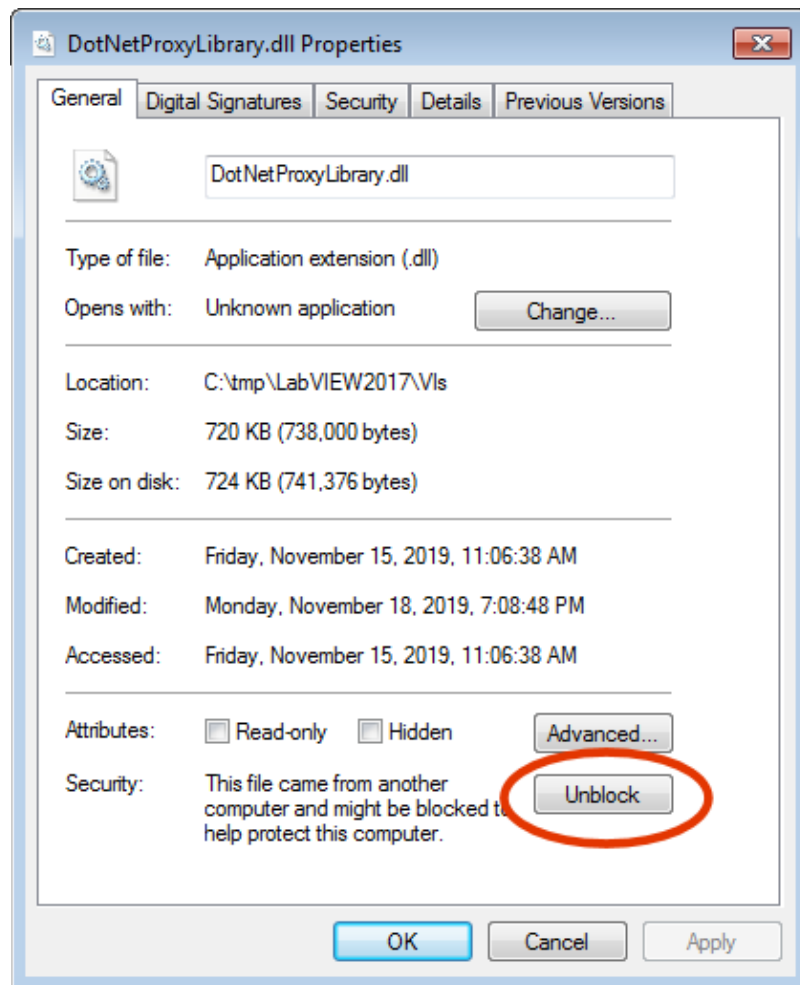
Unblock the DotNetProxyLibrary DLL.

Alternatively, one can modify the LabVIEW configuration by creating, in the same directory as the labview.exe executable, an XML file called *labview.exe.config* containing the following code:

```
<?xml version ="1.0"?>
<configuration>
 <runtime>
 <loadFromRemoteSources enabled="true" />
 </runtime>
</configuration>
```

Make sure to select the correct directory depending on the LabVIEW version you are using (32 bits vs. 64 bits). You can find more information about this file on the National Instruments web site.[2]

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

Method 1 : "Take-out" installation

The simplest way to use the Yoctopuce library is to copy the content of the *VIs* directory wherever you want and to use the VIs in LabVIEW with a simple drag-n-drop operation.

To use the examples provided with the API, it is simpler if you add the directory of Yoctopuce VIs into the list of where LabVIEW must look for VIs that it has not found. You can access this list through the *Tools > Options > Paths > VI Search Path* menu.

---

[2] https://knowledge.ni.com/KnowledgeArticleDetails?id=kA00Z000000P8XnSAK
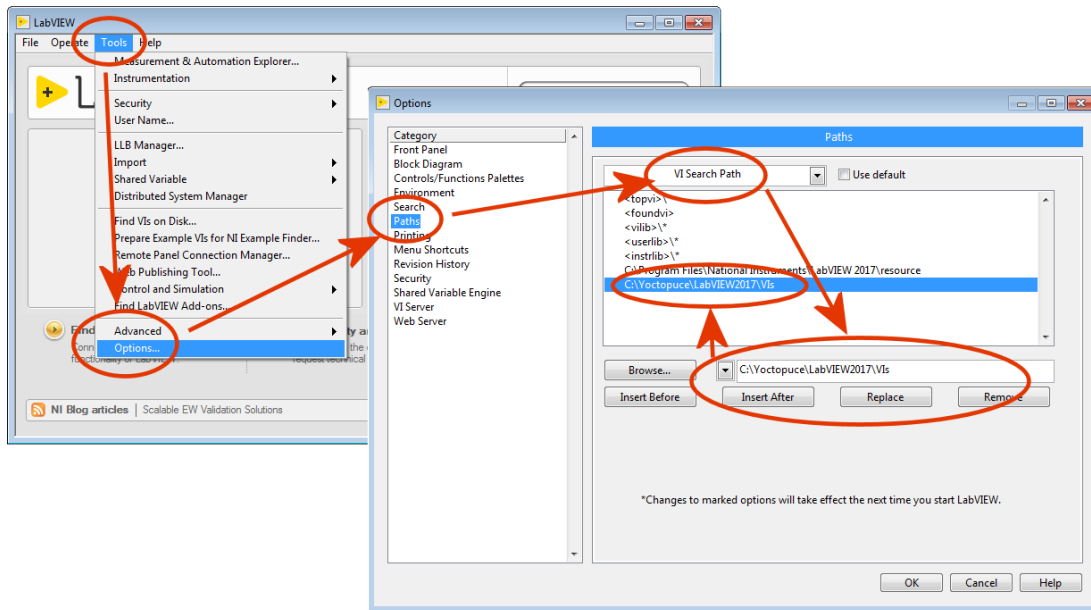
Configuring the "VI Search Path"

Method 2 : Provided installer

In each LabVIEW folder of the Library, you will find a VI named "*Install.vi*", just open the one matching your LabVIEW version.
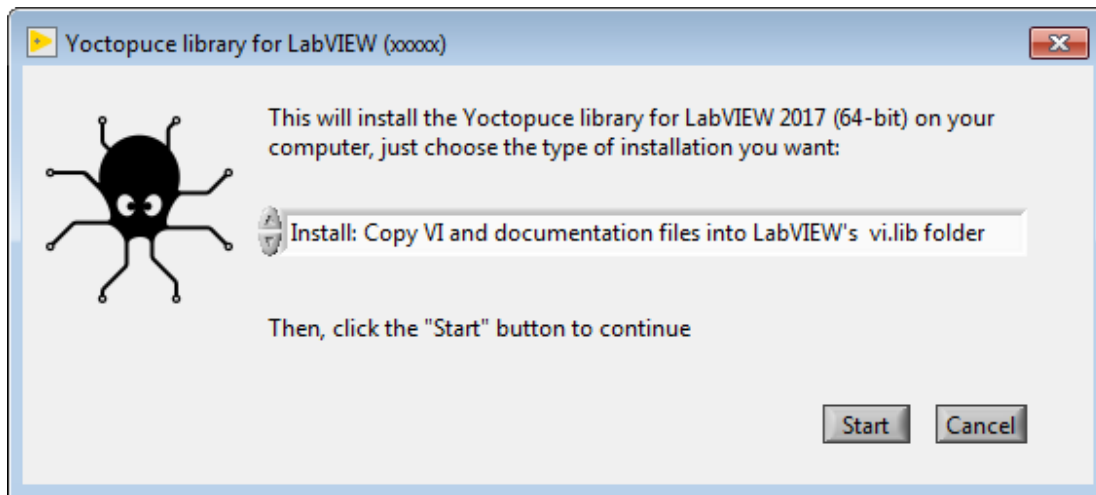
The provider installer

This installer provide 3 installation options:

Install: Keep VI and documentation files where they are.
With this option, VI files are keep in the place where the library has been unzipped. So you will have to make sure these files are not deleted as long as you need them. Here is what the installer will do if that option is chosen:

• All references to Yoctopuce any library paths will be removed from the *viSearchPath* option in the *labview.ini* file.
• A dir.mnu palette file referring to VIs in the install folder will be created in
   C:\Program Files xx\National Instruments\LabVIEW 20xx\vi.lib\addons\Yoctopuce
• A reference to the VIs source install path will be inserted into the *viSearchPath* option in the *labview.ini* file.

Install: Copy VI and documentation files into LabVIEW's vi.lib folder
In that case all required files are copied inside the LabVIEW's installation folder, so you will be able to delete the installation folder once the original installation is complete. Note that programming examples won't be copied. Here is the exact behaviour of the installer in that case:

- All references to Yoctopuce library paths will be removed from *viSearchPath* in *labview.ini* file
- All VIs, DLLs, and documentation files will be copied into:
  C:\Program Files xx\National Instruments\LabVIEW 20xx\vi.lib\Yoctopuce
- VIs will be patched with the path to copied documentation files
- A dir.mnu palette file referring to copied VIs will be created in
  C:\Program Files xx\National Instruments\LabVIEW 20xx\vi.lib\addons\Yoctopuce

Uninstall Yoctopuce Library
this option is meant to remove the LabVIEW library from your LabVIEW installation, here is how it is done:

- All references to Yoctopuce library paths will be removed from *viSearchPath* in *labview.ini* file
- Following folders, if exists, will be removed:
  C:\Program Files xx\National Instruments\LabVIEW 20xx\vi.lib\addons\Yoctopuce
  C:\Program Files xx\National Instruments\LabVIEW 20xx\vi.lib\Yoctopuce

In any case, if the *labview.ini* file needs to be modified, a backup copy will be made beforehand.

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

Once the installation is complete, a Yoctopuce palette will appear in *Functions/Addons* menu.

Method 3 : Installation in a LabVIEW palette (ancillary method)

The steps to manually install the VIs directly in the LabVIEW palette are somewhat more complex. You can find the detailed procedure on the National Instruments web site [3], but here is a summary:

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

---
[3] https://forums.ni.com/t5/Developer-Center-Resources/Creating-a-LabVIEW-Palette/ta-p/3520557
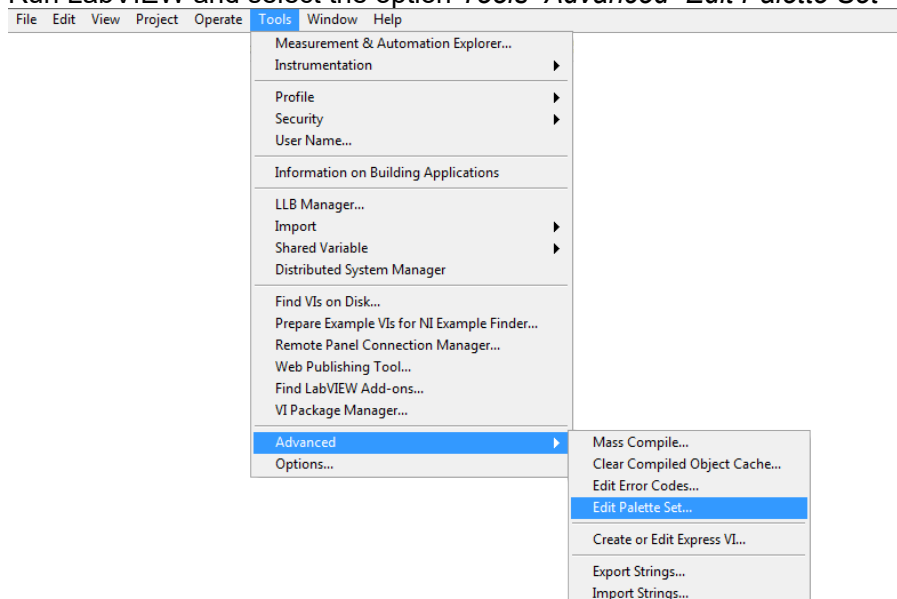
Three windows pop up:

- "Edit Controls and Functions Palette Set"
- "Functions"
- "Controls"
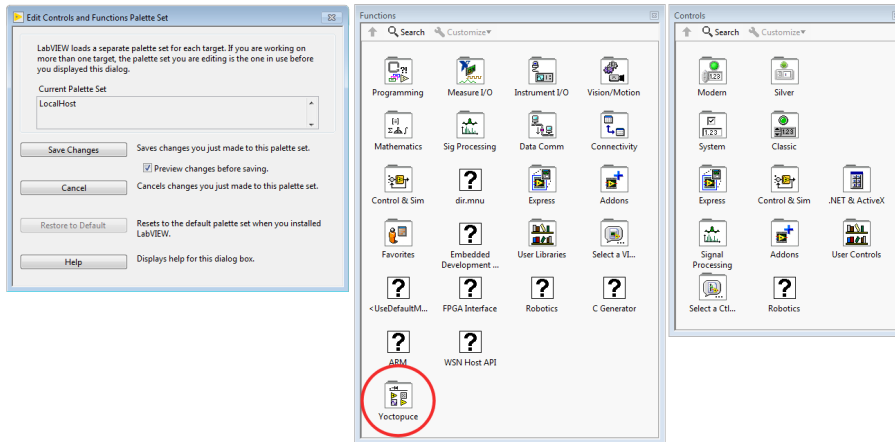
.
In the *Function* window, there is a *Yoctopuce* icon. Double-click it to create an empty "Yoctopuce" window.

4. In the Yoctopuce window, perform a *Right click>Insert>Vi(s)..*
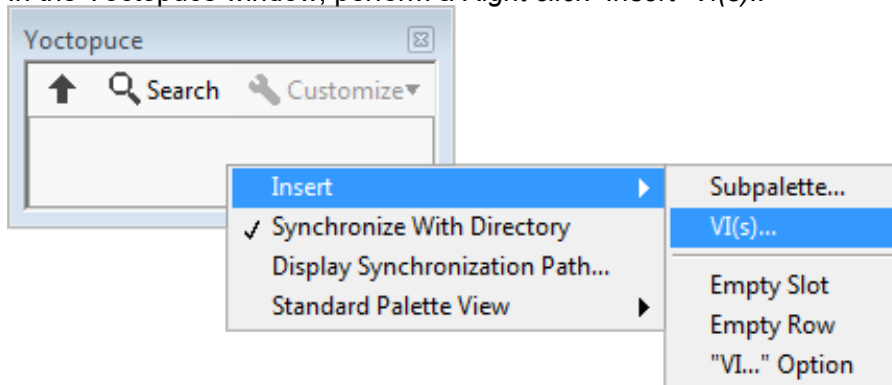
in order to open a file chooser. Put the file chooser in the *vi.lib\Yoctopuce\API* directory that you have created in step 1 and click on *Current Folder*
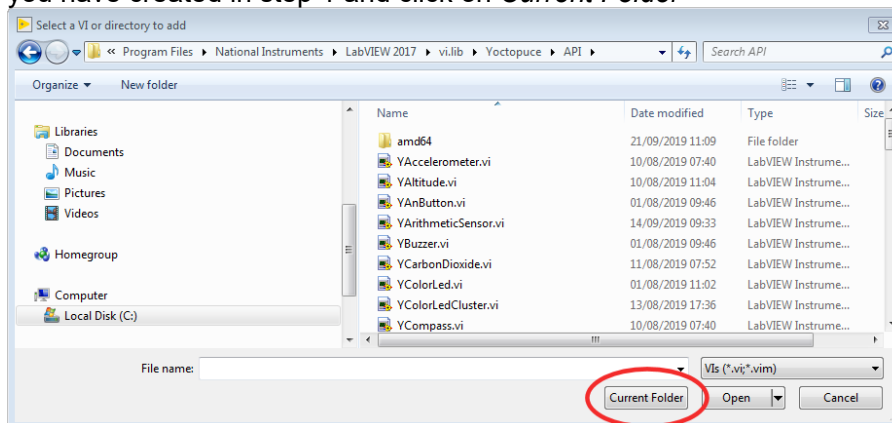
All the Yoctopuce VIs now appear in the Yoctopuce window. By default, they are sorted by alphabetical order, but you can arrange them as you see fit by moving them around with the mouse. For the palette to be easy to use, we recommend to reorganize the icons over 8 columns.

5. In the "*Edit Controls and Functions Palette Set*" window, click on the "Save Changes" button, the window indicates that it has created a `dir.mnu` file in your `Documents` directory.
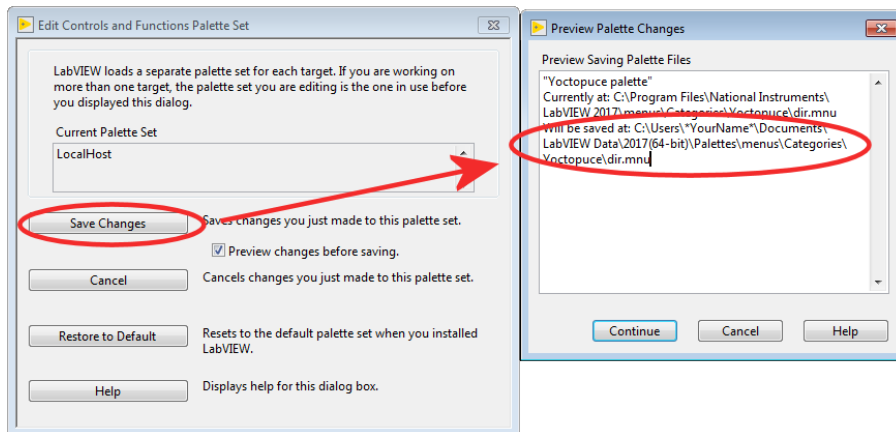
Copy this file in the "`menus\Categories\Yoctopuce`" directory that you have created in step 2.

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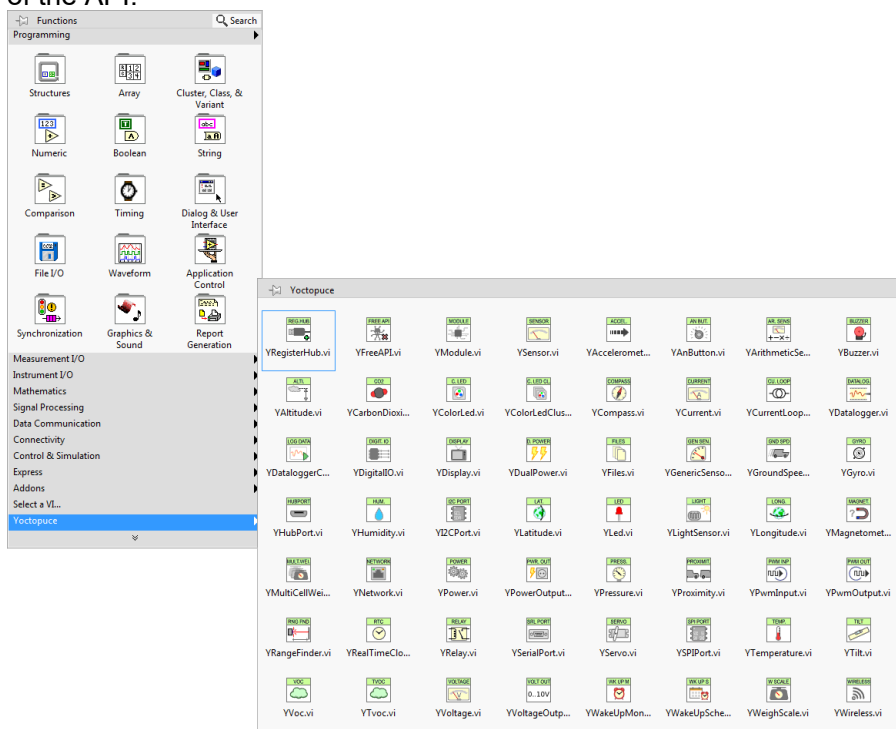
10.4. Presentation of Yoctopuce VIs

The LabVIEW Yoctopuce library contains one VI per class of the Yoctopuce API, as well as a few special VIs. All the VIs have the traditional connectors *Error IN* and *Error Out*.

YRegisterHub

The `YRegisterHub` VI is used to initialize the API. You must imperatively call this VI once before you do anything in relation with Yoctopuce modules.

The YRegisterHub VI
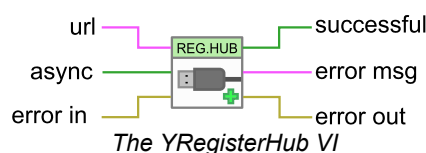
The `YRegisterHub` VI takes a *url* parameter which can be:

- The `"usb"` character string to indicated that you wish to work with local modules, directly connected by USB
- An IP address to indicate that you wish to work with modules which are available through a network connection. This IP address can be that of a YoctoHub[4] or even that of a machine on which the VirtualHub[5] application is running.

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

If everything went well, the *successful* output contains the value TRUE. In the opposite case, it contains the value FALSE and the *error msg* output contains a string of characters with a description of the error.

You can use several `YRegisterHub` VIs with distinct URLs if you so wish. However, on the same machine, there can be only one process accessing local Yoctopuce modules directly by USB (*url* set to `"usb"`). You can easily work around this limitation by running the VirtualHub software on the local machine and using the `"127.0.0.1"` url.

YFreeAPI

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

error in ———————| FREE API |——————— error out
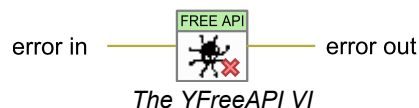
The YFreeAPI VI

You must call the `YFreeAPI` VI when your code is done with the Yoctopuce API. Otherwise, direct USB access (*url* set to `"usb"`) could stay locked after the execution of your VI, and stay so for as long as LabVIEW is not completely closed.

Structure of the VIs corresponding to a class

The other VIs correspond to each function/class of the Yoctopuce API, they all have the same structure:

```
                                    [7] hardware name
                                    [5] is online
    name [11]                       [3] optional reference
    input 1[10]                     [2] output 1
    input 2 [9]                     [1] output 2
    error in [8]                    [0] error out
    create ref [6]
```

Structure of most VIs of the API.
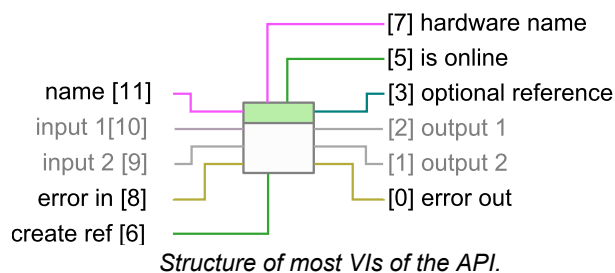
- Connector [11]: *name* must contain the hardware name or the logical name of the intended function.
- Connectors [10] and [9]: input parameters depending on the nature of the VI.
- Connectors [8] and [0] : *error in* and *error out*.
- Connector [7] : Unique hardware name of the found function.
- Connector [5] : *is online* contains TRUE if the function is available, FALSE otherwise.
- Connectors [2] and [1]: output values depending on the nature of the VI.
- Connector [6]: If this input is set to TRUE, connector [3] contains a reference to the *Proxy* objects implemented by the VI[6]. This input is initialized to FALSE by default.

---

[4] www.yoctopuce.com/EN/products/category/extensions-and-networking
[5] http://www.yoctopuce.com/EN/virtualhub.php
[6] see section *Using Proxy objects*

- Connector [3]: Reference on the *Proxy* object implemented by the VI if input [6] is TRUE. This object enables you to access additional features.

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

If the desired function (parameter *name*) is not available, this does not generate an error, but the *is online* output contains FALSE and all the other outputs contain the value "N/A" whenever possible. If the desired function becomes available later in the life of your program, *is online* switches to TRUE automatically.

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

The YModule VI

The `YModule` VI enables you to interface with the "module" section of each Yoctopuce module. It enables you to drive the module led and to know the serial number of the module.
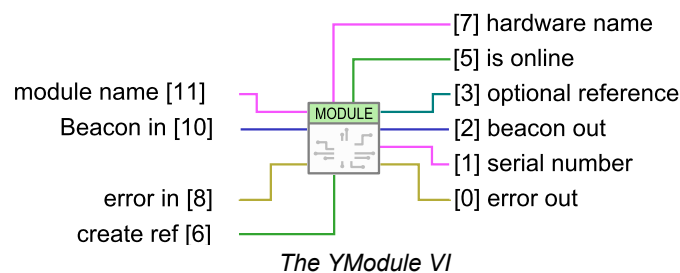
The YModule VI

The *name* input works slightly differently from other VIs. If it is called with a *name* parameter corresponding to a function name, the `YModule` VI finds the *Module* function of the module hosting the function. You can therefore easily find the serial number of the module of any function. This enables you to build the name of other functions which are located on the same module. The following example finds the first available *YHumidity* function and builds the name of the *YTemperature* function located on the same module. The examples provided with the Yoctopuce API make extensive use of this technique.

Using the YModule VI to retrieve functions hosted on the same module
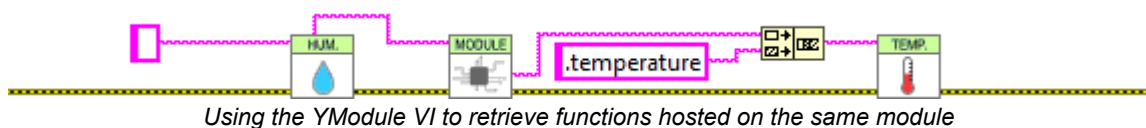
The sensor VIs

All the VIs corresponding to Yoctopuce sensors have exactly the same geometry. Both outputs enable you to retrieve the value measured by the corresponding sensor as well the unit used.

The sensor VIs have all exactly the same geometry
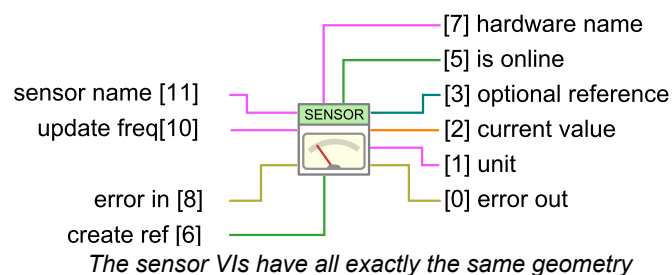
The *update freq* input parameter is a character string enabling you to configure the way in which the output value is updated:

- "auto" : The VI value is updated as soon as the sensor detects a significant modification of the value. It is the default behavior.
- "x/s": The VI value is updated *x* times per second with the current value of the sensor.

- "x/m","x/h": The VI value is updated *x* times per minute (resp. hour) with the average value over the latest period. Note, maximum frequencies are (60/m) and (3600/h), for higher frequencies use the (x/s) syntax.

The update frequency of the VI is a parameter managed by the physical Yoctopuce module. If several VIs try to change the frequency of the same sensor, the valid configuration is that of the latest call. It is however possible to set different update frequencies to different sensors on the same Yoctopuce module.
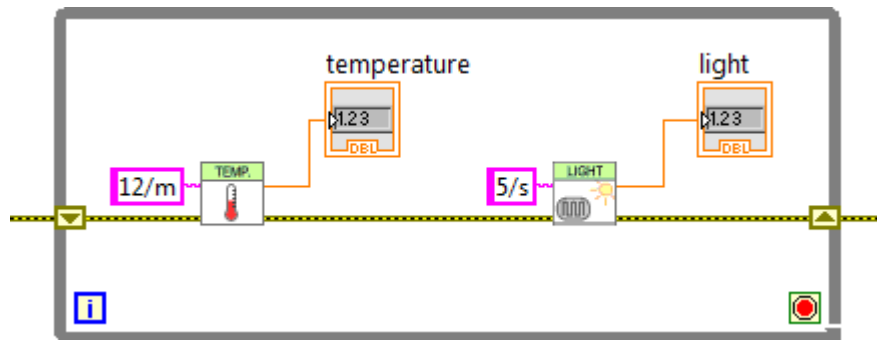
Changing the update frequency of the same module

The update frequency of the VI is completely independent from the sampling frequency of the sensor, which you usually cannot modify. It is useless and counterproductive to define an update frequency higher than the sensor sampling frequency.

10.5. Functioning and use of VIs

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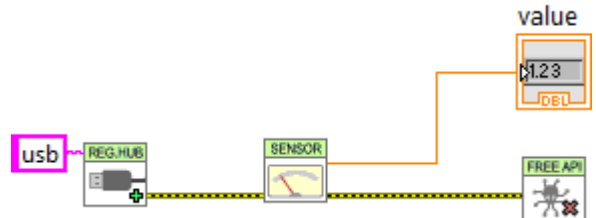
Minimal example of use of the LabVIEW Yoctopuce API

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

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

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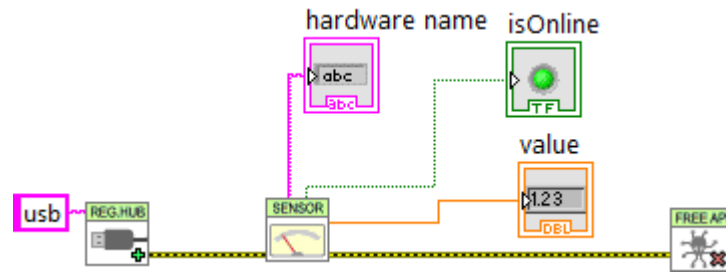
Use of the hardware name and isOnline outputs

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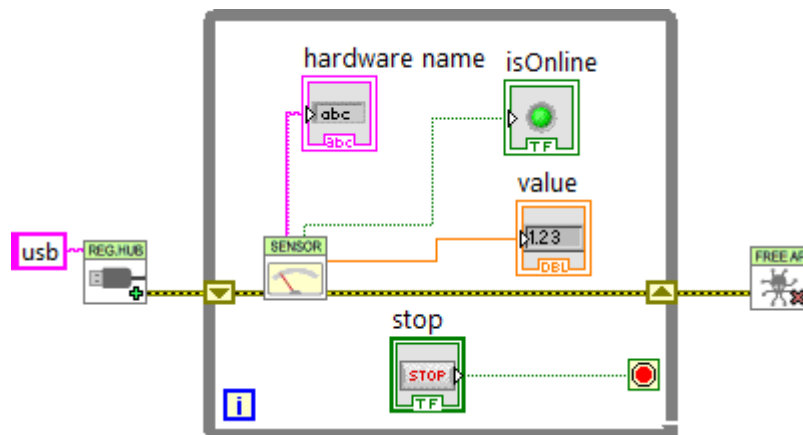
The Yoctopuce VIs can be used in a non-delayed loop

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

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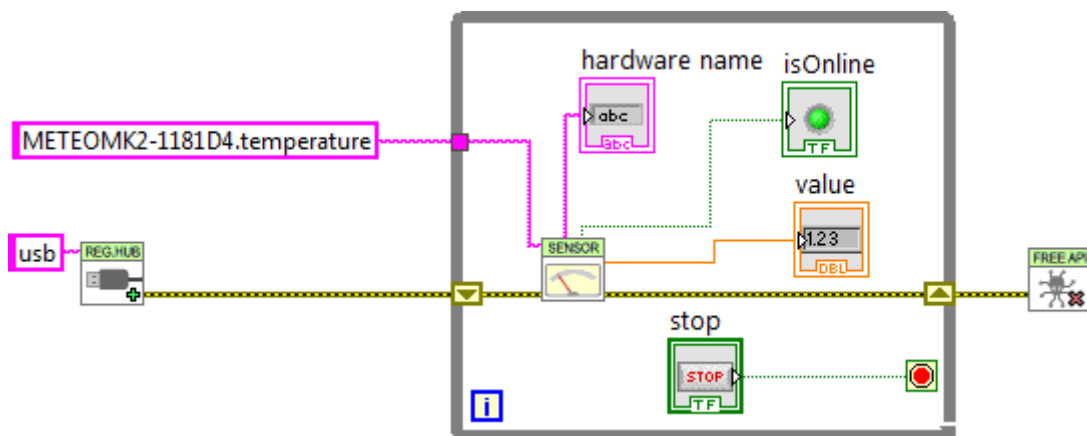
Using names to identify the functions to be used

Error handling

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

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

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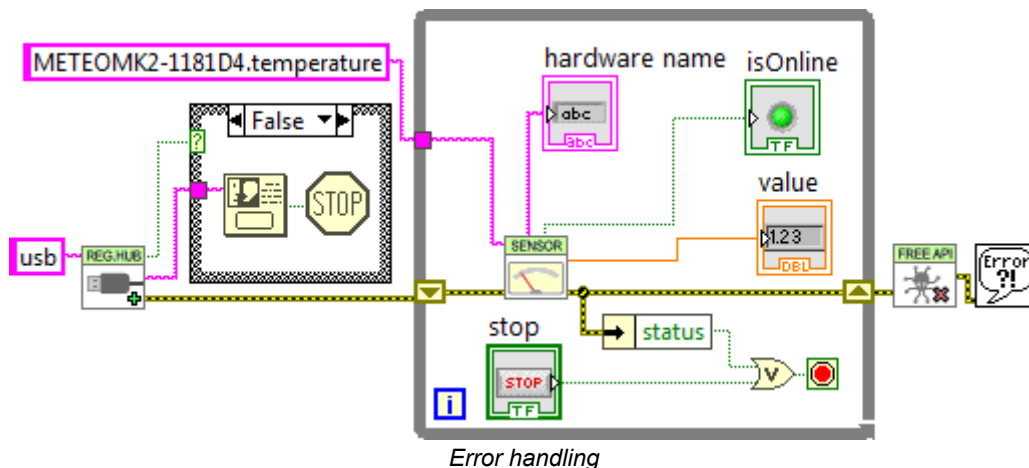
The most common error message is "*Another process is already using yAPI*". It means that another application, LabVIEW or other, already uses the API in native USB mode. For technical reasons, the native USB API can be used by only one application at the same time on the same machine. You can easily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

10.6. Using *Proxy* objects

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

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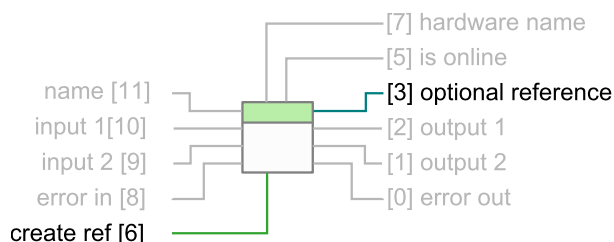
The connectors to obtain a reference on the Proxy object corresponding to the VI

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

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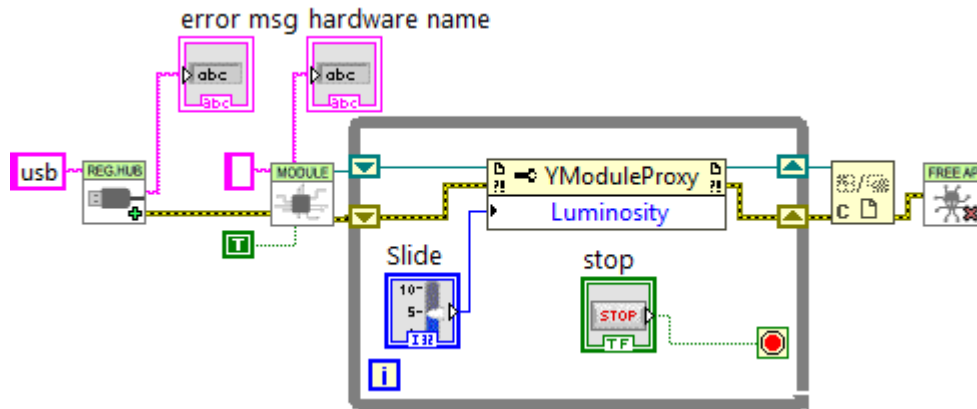
Regulating the luminosity of the leds of a module

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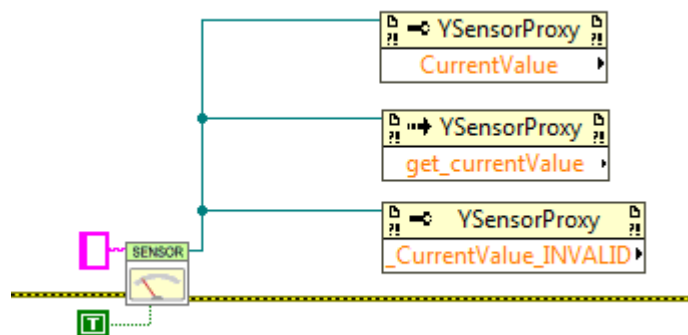
Property and Invoke nodes: Using properties, methods and constants

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

Network mode

On a given machine, there can be only one process accessing local Yoctopuce modules directly by USB (url set to `"usb"`). It is however possible that multiple process connect in parallel to YoctoHubs[7] or tp a machine on which *VirtualHub*[8] is running, including the local machine. Therefore, if you use the local address of your machine (127.0.0.1) and if a VirtualHub runs on it, you can work around the limitation which prevents using the native USB API in parallel.
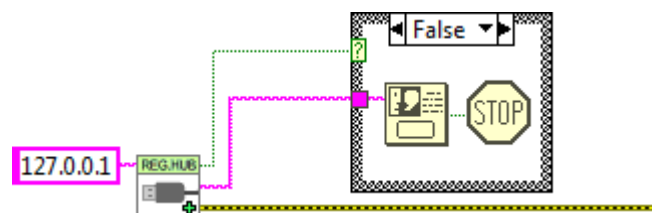
Network mode

In the same way, there is no limitation on the number of network interfaces to which the API can connect itself in parallel. This means that it is quite possible to make multiple calls to the `YRegisterHub` VI. This is the only case where it is useful to call the `YRegisterHub` VI several times in the life of the application.

You can have multiple network connections
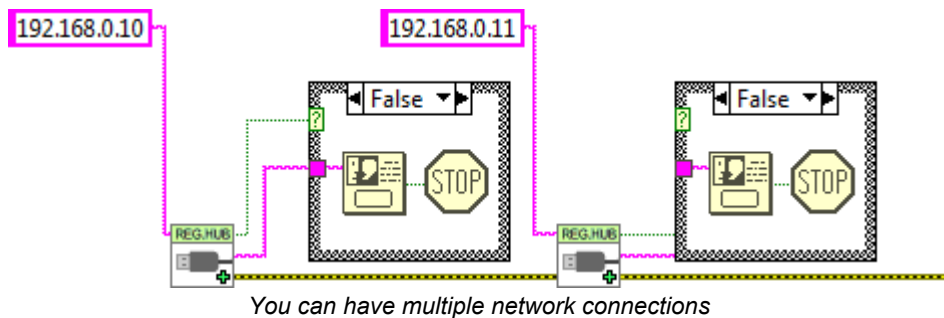
By default, the `YRegisterHub` VI tries to connect itself on the address given as parameter and generates an error (*success*=FALSE) when it cannot do so because nobody answers. But if the *async* parameter is initialized to TRUE, no error is generated when the connection does not succeed. If the connection becomes possible later in the life of the application, the corresponding modules are automatically made available.

Asynchronous connection
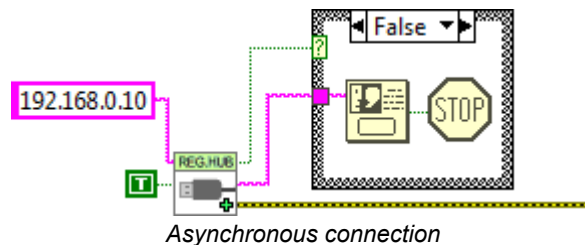
10.7. Managing the data logger

Almost all the Yoctopuce sensors have a data logger which enables you to store the measures of the sensors in the non-volatile memory of the module. You can configure the data logger with the VirtualHub, but also with a little bit of LabVIEW code.

Logging

To do so, you must configure the logging frequency by using the "LogFrequency" property which you can reach with a reference on the *Proxy* object of the sensor you are using. Then, you must turn the data logger on thanks to the `YDataLogger` VI. Note that, like with the `YModule` VI, you can obtain the `YDataLogger` VI corresponding to a module with its own name, but also with the name of any of the functions available on the same module.

Activating the data logger
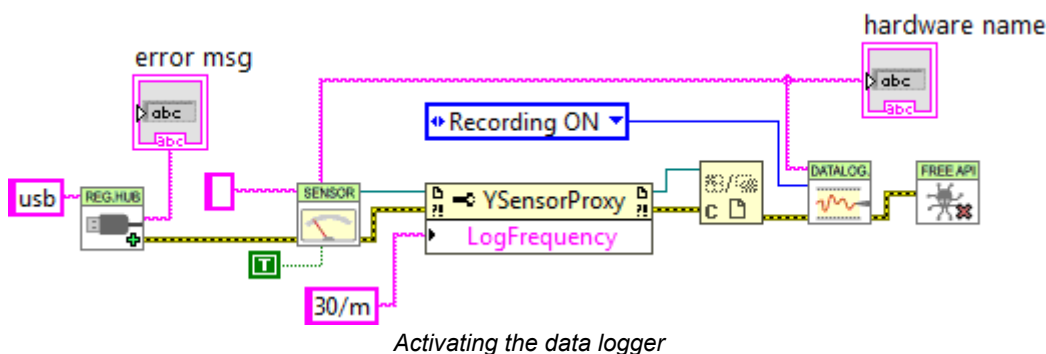
Reading

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

The YDataLoggerContents VI
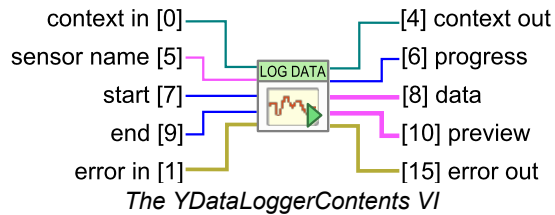
Retrieving the data from the logger of a Yoctopuce module is a slow process which can take up to several tens of seconds. Therefore, we designed the VI enabling this operation to work iteratively.

As a first step, you must call the VI with a sensor name, a start date, and an end date (UTC UNIX timestamp). The (0,0) pair enables you to obtain the complete content of the data logger. This first call enables you to obtain a summary of the data logger content and a context.

As a second step, you must call the *YDataLoggerContents* VI in a loop with the context parameter, until the *progress* output reaches the 100 value. At this time, the data output represents the content of the data logger.

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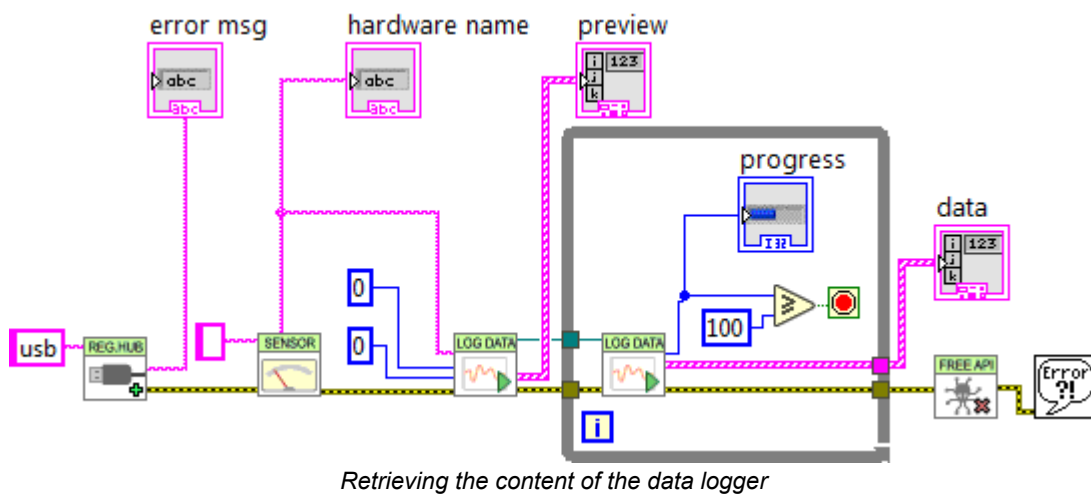
The results and the summary are returned as an array of structures containing the following fields:

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

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

10.8. 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.
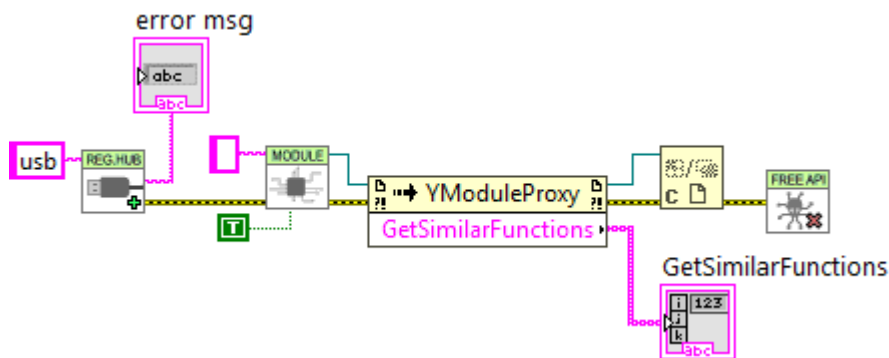
Retrieving the list of all the modules which are connected

10.9. 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.
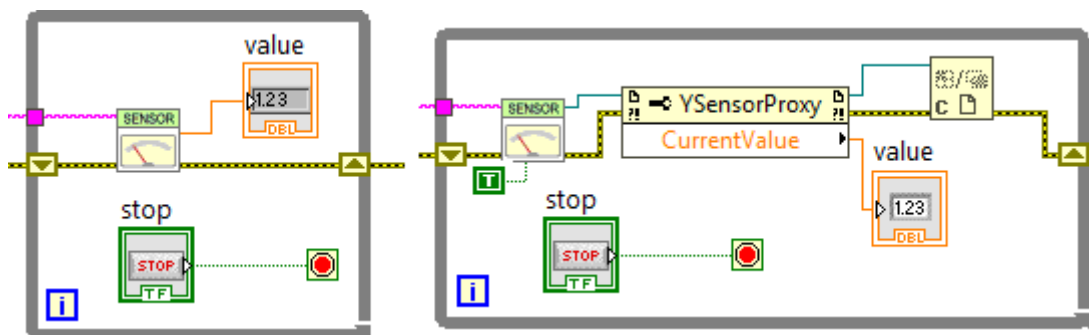
These two loops generate little USB communication and do not need to be slowed down

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.
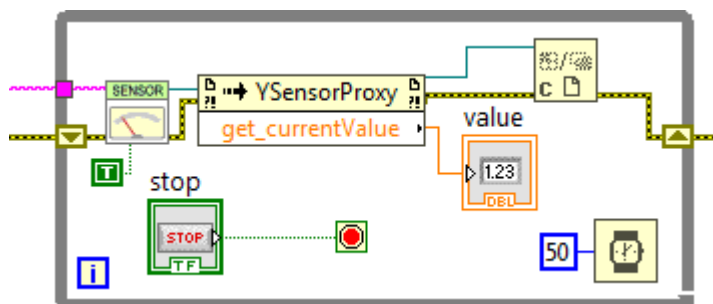
This loop, using a method, must be slowed down

10.10. A full example of a LabVIEW program

Here is a full example of how to use the Yocto-LatchingRelay-C in LabVIEW. After a call to the *RegisterHub* VI, the *YRelay* VI finds the first available relay and displays its state. A switch allows to toggle the relay state. When the application is about to shut down, it frees the Yoctopuce API, thanks to the *YFreeAPI* VI.
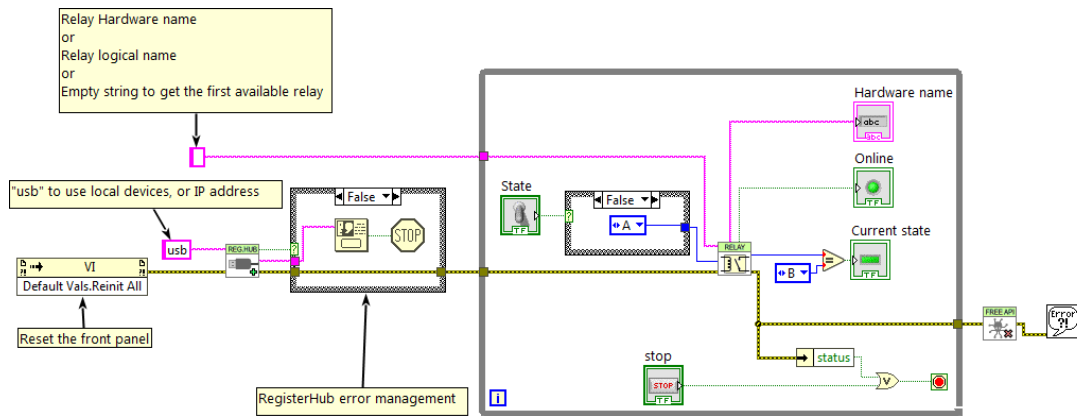
Example of Yocto-LatchingRelay-C usage in LabVIEW.

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

10.11. 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.

11. Using the Yocto-LatchingRelay-C with Java

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

11.1. Getting ready

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

- The Java programming library[1]
- VirtualHub[2] for Windows, macOS or Linux, depending on your OS

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

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

11.2. Control of the Relay function

A few lines of code are enough to use a Yocto-LatchingRelay-C. Here is the skeleton of a Java code snippet to use the Relay 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
relay = YRelay.FindRelay("LTCHRL2C-123456.relay1");

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

---

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

```
    // Use relay.set_state()
    [...]
}

[...]
```

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.

YRelay.FindRelay

The `YRelay.FindRelay` function allows you to find a relay 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-LatchingRelay-C module with serial number *LTCHRL2C-123456* which you have named "*MyModule*", and for which you have given the *relay1* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
relay = YRelay.FindRelay("LTCHRL2C-123456.relay1")
relay = YRelay.FindRelay("LTCHRL2C-123456.MyFunction")
relay = YRelay.FindRelay("MyModule.relay1")
relay = YRelay.FindRelay("MyModule.MyFunction")
relay = YRelay.FindRelay("MyFunction")
```

`YRelay.FindRelay` returns an object which you can then use at will to control the relay.

isOnline

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

set_state

The `set_state()` method of the objet returned by `YRelay.FindRelay` switches the relay position to one of its two outputs. The two possible parameter values are `YRelay.STATE_A` for output A, and `YRelay.STATE_B` for output B.

A real example

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

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

```java
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);
        }

        YRelay relay;
        if (args.length > 0) {
            relay = YRelay.FindRelay(args[0]);
        } else {
```

```
            relay = YRelay.FirstRelay();
            if (relay == null) {
                System.out.println("No module connected (check USB cable)");
                System.exit(1);
            }
        }

        try {

            System.out.println("Switch relay to B");
            relay.set_state(YRelay.STATE_B);
            YAPI.Sleep(1000);
            System.out.println("Switch relay to A");
            relay.set_state(YRelay.STATE_A);
        } catch (YAPI_Exception ex) {
            System.out.println("Module "+relay.describe()+" not connected (check
identification and USB cable)");
        }
        YAPI.FreeAPI();
    }
}
```

11.3. 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.

```java
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.

```java
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();
    }
}
```

11.4. 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.

12. Using the Yocto-LatchingRelay-C with Android

To tell the truth, Android is not a programming language, it is an operating system developed by Google for mobile appliances such as smart phones and tablets. But it so happens that under Android everything is programmed with the same programming language: Java. Nevertheless, the programming paradigms and the possibilities to access the hardware are slightly different from classical Java, and this justifies a separate chapter on Android programming.

12.1. Native access and VirtualHub

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

12.2. Getting ready

Go to the Yoctopuce web site and download the Java for Android programming library[1]. The library is available as source files, and also as a jar file. Connect your modules, decompress the library files in the directory of your choice, and configure your Android programming environment so that it can find them.

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

12.3. Compatibility

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

Android version

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

---

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

USB *host* support

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

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

12.4. Activating the USB port under Android

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

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

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

When first accessing a Yoctopuce module, Android opens a window to inform the user that the application is going to access the connected module. The user can deny or authorize access to the device. If the user authorizes the access, the application can access the connected device as long as it stays connected. To enable the Yoctopuce library to correctly manage these authorizations, your must provide a pointer on the application context by calling the EnableUSBHost method of the YAPI class before the first USB access. This function takes as arguments an object of the `android.content.Context` class (or of a subclass). As the Activity class is a subclass of `Context`, it is simpler to call `YAPI.EnableUSBHost(this);` in the method `onCreate` of your application. If the object passed as parameter is not of the correct type, a `YAPI_Exception` exception is generated.

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

Autorun

It is possible to register your application as a default application for a USB module. In this case, as soon as a module is connected to the system, the application is automatically launched. You must

add *<action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"/>* in the section *<intent-filter>* of the main activity. The section *<activity>* must have a pointer to an XML file containing the list of USB modules which can run the application.

```xml
<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
<?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>
```

12.5. Control of the Relay function

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

```java
[...]
// Enable detection of USB devices
YAPI.EnableUSBHost(this);
YAPI.RegisterHub("usb");
[...]
// Retrieve the object used to interact with the device
relay = YRelay.FindRelay("LTCHRL2C-123456.relay1");

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

[...]
```

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.

YRelay.FindRelay

The `YRelay.FindRelay` function allows you to find a relay 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-LatchingRelay-C module with serial number *LTCHRL2C-123456* which you have named "*MyModule*", and for which you have given the *relay1* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
relay = YRelay.FindRelay("LTCHRL2C-123456.relay1")
relay = YRelay.FindRelay("LTCHRL2C-123456.MyFunction")
relay = YRelay.FindRelay("MyModule.relay1")
relay = YRelay.FindRelay("MyModule.MyFunction")
relay = YRelay.FindRelay("MyFunction")
```

`YRelay.FindRelay` returns an object which you can then use at will to control the relay.

isOnline

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

set_state

The `set_state()` method of the objet returned by `YRelay.FindRelay` switches the relay position to one of its two outputs. The two possible parameter values are `YRelay.STATE_A` for output A, and `YRelay.STATE_B` for output B.

A real example

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

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

```java
package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemSelectedListener;
import android.widget.ArrayAdapter;
import android.widget.Spinner;

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

public class GettingStarted_Yocto_LatchedRelay extends Activity implements
OnItemSelectedListener
{

    private YRelay relay = null;
    private ArrayAdapter<String> aa;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.gettingstarted_yocto_latchedrelay);
```

```java
        Spinner my_spin = (Spinner) findViewById(R.id.spinner1);
        my_spin.setOnItemSelectedListener(this);
        aa = new ArrayAdapter<String>(this, android.R.layout.simple_spinner_item);
        aa.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
        my_spin.setAdapter(aa);
    }

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

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

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

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

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

    /** Called when the user touches the button State A */
    public void setStateA(View view)
    {
        // Do something in response to button click
        if (relay != null)
            try {
                relay.setState(YRelay.STATE_A);
            } catch (YAPI_Exception e) {
                e.printStackTrace();
            }
    }

    /** Called when the user touches the button State B */
    public void setStateB(View view)
    {
        // Do something in response to button click
        if (relay != null)
            try {
                relay.setState(YRelay.STATE_B);
            } catch (YAPI_Exception e) {
                e.printStackTrace();
            }
    }
}
```

12.6. 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.

```java
package com.yoctopuce.doc_examples;

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

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

public class ModuleControl extends Activity implements OnItemSelectedListener
{

    private ArrayAdapter<String> aa;
    private YModule module = null;

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

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

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

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

    private void DisplayModuleInfo()
    {
        TextView field;
        if (module == null)
            return;
        try {
            field = (TextView) findViewById(R.id.serialfield);
            field.setText(module.getSerialNumber());
```

```
            field = (TextView) findViewById(R.id.logicalnamefield);
            field.setText(module.getLogicalName());
            field = (TextView) findViewById(R.id.luminosityfield);
            field.setText(String.format("%d%%", module.getLuminosity()));
            field = (TextView) findViewById(R.id.uptimefield);
            field.setText(module.getUpTime() / 1000 + " sec");
            field = (TextView) findViewById(R.id.usbcurrentfield);
            field.setText(module.getUsbCurrent() + " mA");
            Switch sw = (Switch) findViewById(R.id.beaconswitch);
            sw.setChecked(module.getBeacon() == YModule.BEACON_ON);
            field = (TextView) findViewById(R.id.logs);
            field.setText(module.get_lastLogs());

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

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

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

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

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

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

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

Changing the module settings

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

```
package com.yoctopuce.doc_examples;

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

```java
import android.widget.AdapterView.OnItemSelectedListener;
import android.widget.ArrayAdapter;
import android.widget.EditText;
import android.widget.Spinner;
import android.widget.TextView;
import android.widget.Toast;

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

public class SaveSettings extends Activity implements OnItemSelectedListener
{

    private ArrayAdapter<String> aa;
    private YModule module = null;

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

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

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

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

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

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

```
    }

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

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

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

}
```

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

Listing the modules

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

```
package com.yoctopuce.doc_examples;

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

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

public class Inventory extends Activity
{

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

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

        try {
            YAPI.UpdateDeviceList();
```

```
            YModule module = YModule.FirstModule();
            while (module != null) {
                String line = module.get_serialNumber() + " (" + module.get_productName() +
")";
                TextView tx = new TextView(this);
                tx.setText(line);
                tx.setTextSize(TypedValue.COMPLEX_UNIT_SP, 20);
                layout.addView(tx);
                module = module.nextModule();
            }
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
    }

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

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

}
```

12.7. 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.

13. Using Yocto-LatchingRelay-C with TypeScript

TypeScript is an enhanced version of the JavaScript programming language. It is a syntaxic 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 writting, 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`.

13.1. Using the Yoctopuce library for TypeScript

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

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

```
npm install -g typescript
```

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

- The TypeScript programming library[1]
- The VirtualHub software[2] for Windows, macOS, or Linux, depending on your OS. TypeScript and JavaScript are part of those languages which do not generally allow you to directly access to USB peripherals. Therefore the library can only be used to access network-enabled devices (connected through a YoctoHub), or USB devices accessible through Yoctopuce TCP/IP to USB gateway, named *VirtualHub*. No extra driver will be needed, though.

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

```
npm install
```

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

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

13.2. 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 by *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.

13.3. Control of the Relay function

A few lines of code are enough to use a Yocto-LatchingRelay-C. Here is the skeleton of a TypeScript code snipplet to use the Relay function.

```typescript
// 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 { YRelay } from 'yoctolib-cjs/yocto_relay.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 relay: YRelay = YRelay.FindRelay("LTCHRL2C-123456.relay1");

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

Let us look at these lines in more details.

yocto_api and yocto_relay import

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

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

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

YAPI.RegisterHub

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

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

YRelay.FindRelay

The `FindRelay` method allows you to find a relay 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-LatchingRelay-C module with serial number *LTCHRL2C-123456* which you have named "*MyModule*", and for which you have given the *relay1* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
relay = YRelay.FindRelay("LTCHRL2C-123456.relay1")
relay = YRelay.FindRelay("LTCHRL2C-123456.MaFonction")
relay = YRelay.FindRelay("MonModule.relay1")
relay = YRelay.FindRelay("MonModule.MaFonction")
relay = YRelay.FindRelay("MaFonction")
```

`YRelay.FindRelay` returns an object which you can then use at will to control the relay.

isOnline

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

set_state

The `set_state()` method of the objet returned by `YRelay.FindRelay` switches the relay position to one of its two outputs. The two possible parameter values are `YRelay.STATE_A` for output A, and `YRelay.STATE_B` for output B.

A real example, for Node.js

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

If your Yocto-LatchingRelay-C is not connected on the host running the browser, replace in the example the address `127.0.0.1` by the IP address of the host on which the Yocto-LatchingRelay-C is connected and where you run the VirtualHub.

```
import { YAPI, YErrorMsg, YModule } from 'yoctolib-cjs/yocto_api_nodejs.js';
import { YRelay } from 'yoctolib-cjs/yocto_relay.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 relay to use
    let target: string;
    if(args[0] == "any") {
        let anyrelay = YRelay.FirstRelay();
        if (anyrelay == null) {
            console.log("No module connected (check USB cable)\n");
            await YAPI.FreeAPI();
            return;
        }
        let module: YModule = await anyrelay.get_module();
        target = await module.get_serialNumber();
    } else {
```

```
        target = args[0];
    }

    // Switch relay as requested
    console.log("Set ouput to " + args[1]);
    let relay: YRelay = YRelay.FindRelay(target + ".relay1");
    if(await relay.isOnline()) {
        await relay.set_output(args[1] == "ON" ? YRelay.OUTPUT_ON : YRelay.OUTPUT_OFF);
    } else {
        console.log("Module not connected (check identification and USB cable)\n");
    }

    await YAPI.FreeAPI();
}

if(process.argv.length < 4) {
    console.log("usage: node demo.js <serial_number> [ ON | OFF ]");
    console.log("       node demo.js <logical_name> [ ON | OFF ]");
    console.log("       node demo.js any [ ON | OFF ]");
} else {
    startDemo(process.argv.slice(process.argv.length - 2));
}
```

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 ar 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-LatchingRelay-C**. You will find there an HTML file named `app.html`, and a TypeScript file `app.ts` similar to the code above, but with a few changes since it has to interact through an HTML page rather than through the JavaScript console.

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

```
npm run app-server
```

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

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

13.4. Control of the module part

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

```typescript
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.

```typescript
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.

```typescript
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();
```

13.5. Error handling

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

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

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

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

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

14. Using Yocto-LatchingRelay-C with JavaScript / EcmaScript

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

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

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

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

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

We don't suggest using `jspm` anymore now that `async` / `await` are part of the standard.

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

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

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

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

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

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

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

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

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

14.2. Using Yoctopuce library for JavaScript / EcmaScript 2017

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

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

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

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

Using the official Yoctopuce library for node.js

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

---

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

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

```
npm install
```

Once done, you can start the example file using:

```
node demo.js
```

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

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

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

Using the Yoctopuce library within a browser (HTML)

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

Using the Yoctoluce library on older JavaScript engines

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

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

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

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

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

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

Backward-compatibility with the old JavaScript library

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

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

by

---

[3] http://babeljs.io

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

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

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

by

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

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

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

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

you can use:

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

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

You can also rewrite this last asynchronous call as:

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

14.3. Control of the Relay function

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

```
// For Node.js, we use function require()
// For HTML, we would use &lt;script src="..."&gt;
require('yoctolib-es2017/yocto_api.js');
require('yoctolib-es2017/yocto_relay.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 relay = YRelay.FindRelay("LTCHRL2C-123456.relay1");

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

```
if(await relay.isOnline())
{
    // Use relay.set_state()
    [...]
}
```

Let us look at these lines in more details.

yocto_api and yocto_relay import

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

YAPI.RegisterHub

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

YRelay.FindRelay

The `FindRelay` method allows you to find a relay 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-LatchingRelay-C module with serial number *LTCHRL2C-123456* which you have named "*MyModule*", and for which you have given the *relay1* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
relay = YRelay.FindRelay("LTCHRL2C-123456.relay1")
relay = YRelay.FindRelay("LTCHRL2C-123456.MaFonction")
relay = YRelay.FindRelay("MonModule.relay1")
relay = YRelay.FindRelay("MonModule.MaFonction")
relay = YRelay.FindRelay("MaFonction")
```

`YRelay.FindRelay` returns an object which you can then use at will to control the relay.

isOnline

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

set_state

The `set_state()` method of the objet returned by `YRelay.FindRelay` switches the relay position to one of its two outputs. The two possible parameter values are `YRelay.STATE_A` for output A, and `YRelay.STATE_B` for output B.

A real example, for Node.js

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

If your Yocto-LatchingRelay-C is not connected on the host running the browser, replace in the example the address `127.0.0.1` with the IP address of the host on which the Yocto-LatchingRelay-C is connected and where you run the VirtualHub.

```
"use strict";
```

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

async function startDemo(args)
{
    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 the relay to use
    let target;
    if(args[0] == "any") {
        let anyrelay = YRelay.FirstRelay();
        if (anyrelay == null) {
            console.log("No module connected (check USB cable)\n");
            process.exit(1);
        }
        let module = await anyrelay.get_module();
        target = await module.get_serialNumber();
    } else {
        target = args[0];
    }

    // Switch relay as requested
    console.log("Set ouput to " + args[1]);
    let relay = YRelay.FindRelay(target + ".relay1");
    if(await relay.isOnline()) {
        await relay.set_output(args[1] == "ON" ? YRelay.OUTPUT_ON : YRelay.OUTPUT_OFF);
    } else {
        console.log("Module not connected (check identification and USB cable)\n");
    }

    await YAPI.FreeAPI();
}

if(process.argv.length < 4) {
    console.log("usage: node demo.js <serial_number> [ ON | OFF ]");
    console.log("       node demo.js <logical_name> [ ON | OFF ]");
    console.log("       node demo.js any [ ON | OFF ]");
} else {
    startDemo(process.argv.slice(process.argv.length - 2));
}
```

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

```
npm install
```

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

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

Same example, but this time running in a browser

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

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
```

```html
  <title>Hello World</title>
  <script src="../../lib/yocto_api.js"></script>
  <script src="../../lib/yocto_relay.js"></script>
  <script>
    let relay;

    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 anyRelay = YRelay.FirstRelay();
        if(anyRelay) {
          let module = await anyRelay.module();
          serial = await module.get_serialNumber();
          document.getElementById('serial').value = serial;
        }
      }
      relay = YRelay.FindRelay(serial+".relay1");
      if(await relay.isOnline()) {
        document.getElementById('msg').value = '';
      } else {
        document.getElementById('msg').value = 'Module not connected';
      }
      setTimeout(refresh, 500);
    }

    window.sw = function(state)
    {
      relay.set_output(state ? YRelay.OUTPUT_ON : YRelay.OUTPUT_OFF);
    };

    startDemo();
  </script>
</head>
<body>
 Module to use: <input id='serial'>
 <input id='msg' style='color:red;border:none;' readonly><br>
 Relay <a href='javascript:sw(0);' >OFF</a> / <a href='javascript:sw(1);'>ON</a><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,

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.

```javascript
"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);
}
```

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-LatchingRelay-C with PHP

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

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

To start your tests with PHP, you need a PHP 7.1 (or more recent) server[1], preferably locally on you machine. If you wish to use the PHP server of your internet provider, it is possible, but you will probably need to configure your ADSL router for it to accept and forward TCP request on the 4444 port.

15.1. Getting ready

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

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

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

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

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

---

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

In the library archive, there are thus three subdirectories:

- php7
- php8
- phpPSR

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

15.2. Control of the Relay function

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

```php
include('yocto_api.php');
include('yocto_relay.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
$relay = YRelay::FindRelay("LTCHRL2C-123456.relay1");

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

Let's look at these lines in more details.

yocto_api.php and yocto_relay.php

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

YAPI::RegisterHub

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

YRelay::FindRelay

The `YRelay::FindRelay` function allows you to find a relay 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-LatchingRelay-C module with serial number *LTCHRL2C-123456* which you have named "*MyModule*", and for which you have given the *relay1* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```php
$relay = YRelay::FindRelay("LTCHRL2C-123456.relay1");
$relay = YRelay::FindRelay("LTCHRL2C-123456.MyFunction");
$relay = YRelay::FindRelay("MyModule.relay1");
$relay = YRelay::FindRelay("MyModule.MyFunction");
$relay = YRelay::FindRelay("MyFunction");
```

`YRelay::FindRelay` returns an object which you can then use at will to control the relay.

isOnline

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

set_state

The `set_state()` method of the objet returned by `yFindRelay` switches the relay position to one of its two outputs. The two possible parameter values are `Y_STATE_A` for output A, and `Y_STATE_B` for output B.

A real example

Open your preferred text editor[4], copy the code sample below, save it with the Yoctopuce library files in a location which is accessible to you web server, then use your preferred web browser to access this page. The code is also provided in the directory **Examples/Doc-GettingStarted-Yocto-LatchingRelay-C** of the Yoctopuce library.

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

```php
<HTML>
<HEAD>
 <TITLE>Hello World</TITLE>
</HEAD>
<BODY>
<FORM method='get'>
<?php
  include('yocto_api.php');
  include('yocto_relay.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
      $relay = YRelay::FindRelay("$serial.relay1");
      if (!$relay->isOnline()) {
          die("Module not connected (check serial and USB cable)");
      }
  } else {
      // or use any connected module suitable for the demo
      $relay = YRelay::FirstRelay();
      if(is_null($relay)) {
          die("No module connected (check USB cable)");
      } else {
          $serial = $relay->module()->get_serialnumber();
      }
  }
  Print("Module to use: <input name='serial' value='$serial'><br>");

  // Drive the selected module
  if (isset($_GET['state'])) {
      $state = $_GET['state'];
      if ($state=='A') $relay->set_state(Y_STATE_A);
      if ($state=='B') $relay->set_state(Y_STATE_B);
  }
  YAPI::FreeAPI();
?>
<input type='radio' name='state' value='A'>Output A
<input type='radio' name='state' value='B'>Output B
<br><input type='submit'>
</FORM>
</BODY>
```

---

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

```
</HTML>
```

15.3. Control of the module part

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

```
<HTML>
<HEAD>
 <TITLE>Module Control</TITLE>
</HEAD>
<BODY>
<FORM method='get'>
<?php
  include('../../php8/yocto_api.php');

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

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

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

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

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

Changing the module settings

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

```php
<HTML>
<HEAD>
 <TITLE>save settings</TITLE>
<BODY>
<FORM method='get'>
<?php
  include('../../php8/yocto_api.php');

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

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

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

  if (isset($_GET['newname'])){
      $newname = $_GET['newname'];
      if (!yCheckLogicalName($newname))
          die('Invalid name');
      $module->set_logicalName($newname);
      $module->saveToFlash();
  }
  printf("Current name: %s<br>", $module->get_logicalName());
  print("New name: <input name='newname' value='' maxlength=19><br>");
  YAPI::FreeAPI();
?>
<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('../../php8/yocto_api.php');
    YAPI::RegisterHub("http://127.0.0.1:4444/");
    $module = YModule::FirstModule();
    while (!is_null($module)) {
        printf("%s (%s)<br>\n", $module->get_serialNumber(),
                $module->get_productName());
        $module=$module->nextModule();
    }
    YAPI::FreeAPI();
    ?>
</TT>
</BODY>
</HTML>
```

15.4. HTTP callback API and NAT filters

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

The NAT filter: advantages and disadvantages

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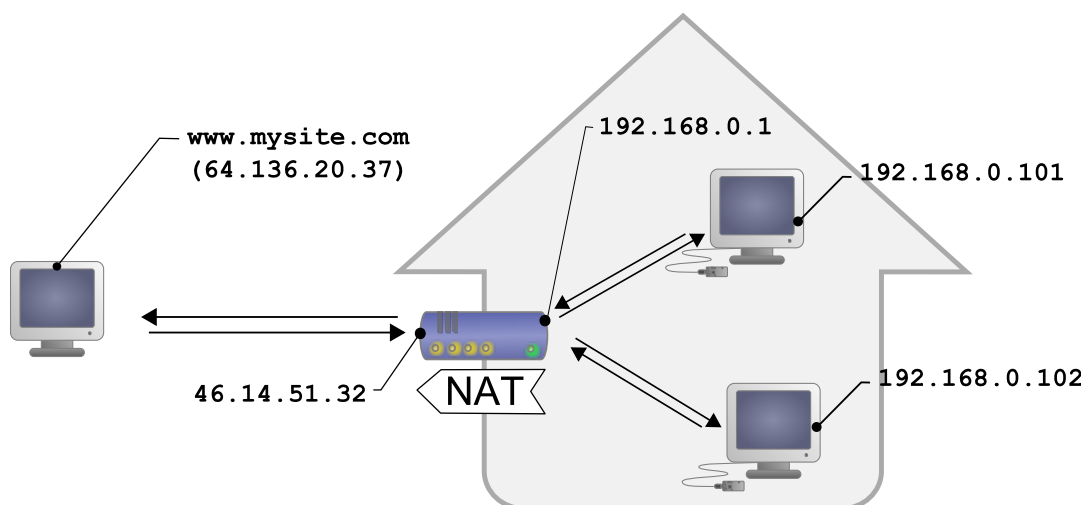
Typical DSL configuration: LAN machines are isolated from the outside by the DSL router

Transposed to the network, we have the following: appliances connected to your home automation network can communicate with one another using a local IP address (of the 192.168.xxx.yyy type), and contact Internet servers through their public address. However, seen from the outside, you have

only one official IP address, assigned to the DSL router only, and you cannot reach your network appliances directly from the outside. It is rather restrictive, but it is a relatively efficient protection against intrusions.

Responses from request from LAN machines are routed.

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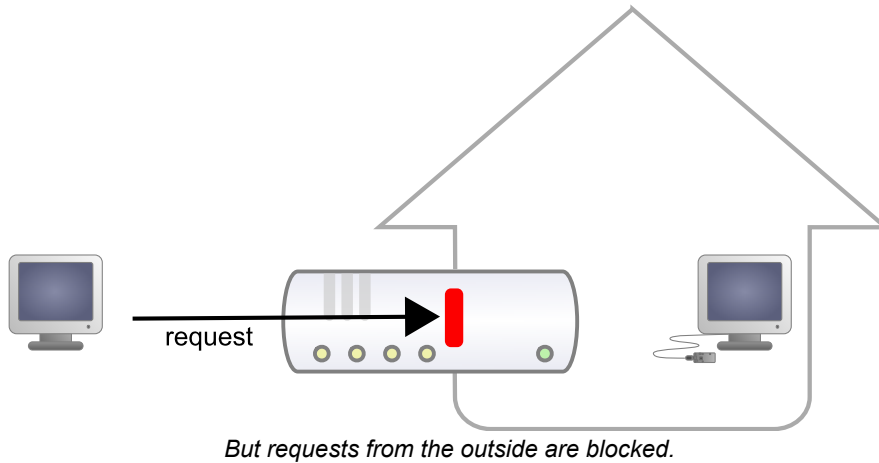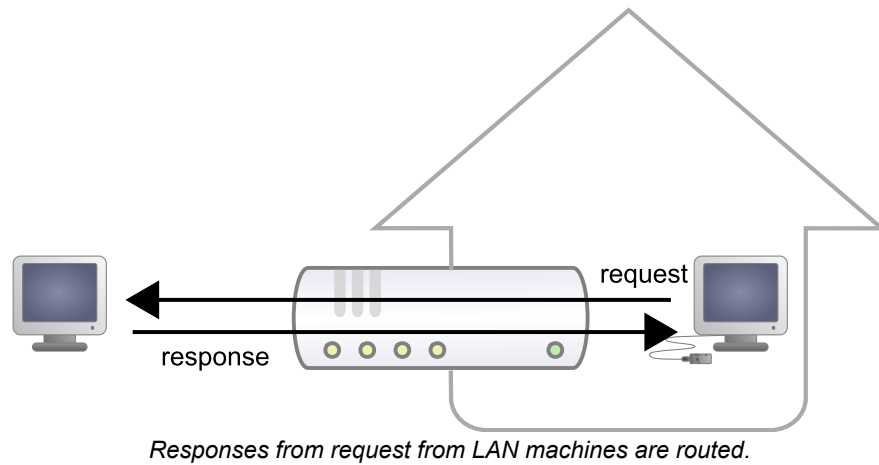
Seeing Internet without being seen provides an enormous security advantage. However, this signifies that you cannot, a priori, set up your own web server at home to control a home automation installation from the outside. A solution to this problem, advised by numerous home automation system dealers,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

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

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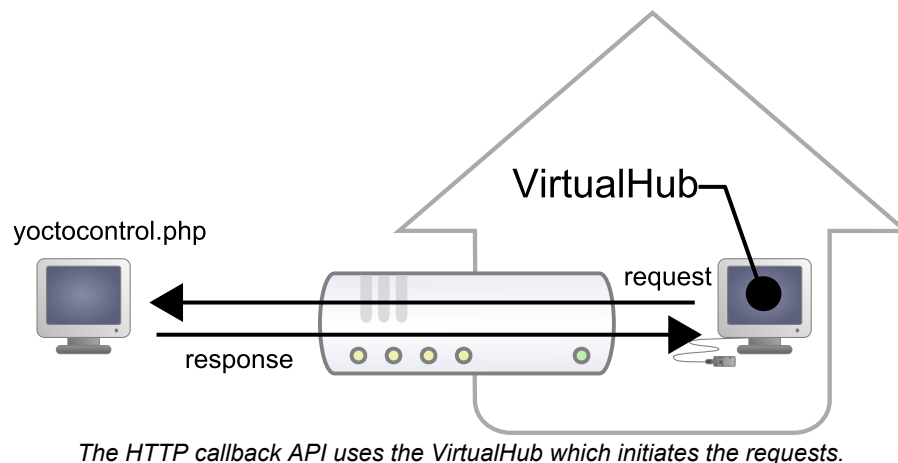
Configuration

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

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

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

Click on the "configure" button on the first line

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

And select "Yocto-API callback".

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

Usage

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

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

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

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

Common issues

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

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

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

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

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

Limitations

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

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

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-LatchingRelay-C with Visual Basic .NET

VisualBasic has long been the most favored entrance path to the Microsoft world. Therefore, we had to provide our library for this language, even if the new trend is shifting to C#. We support Visual Studio 2017 and its more recent versions.

16.1. Installation

Download the Visual Basic Yoctopuce library from the Yoctopuce web site[1]. 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.

16.2. Using the Yoctopuce API in a Visual Basic project

The Visual Basic.NET Yoctopuce library is composed of a DLL and of source files in Visual Basic. The DLL is not a .NET DLL, but a classic DLL, written in C, which manages the low level communications with the modules[2]. The source files in Visual Basic manage the high level part of the API. Therefore, your need both this DLL and the .vb files of the `sources` directory to create a project managing Yoctopuce modules.

Configuring a Visual Basic project

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

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

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

---

[1] www.yoctopuce.com/EN/libraries.php
[2] The sources of this DLL are available in the C++ API

Then add in the same manner the `yapi.dll` DLL, located in the `Sources/dll` directory[3]. 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.

16.3. Control of the Relay function

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

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

' Retrieve the object used to interact with the device
Dim relay As YRelay
relay = YRelay.FindRelay("LTCHRL2C-123456.relay1")

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

YRelay.FindRelay

The `YRelay.FindRelay` function allows you to find a relay 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-LatchingRelay-C module with serial number *LTCHRL2C-123456* which you have named "*MyModule*", and for which you have given the *relay1* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
relay = YRelay.FindRelay("LTCHRL2C-123456.relay1")
relay = YRelay.FindRelay("LTCHRL2C-123456.MyFunction")
relay = YRelay.FindRelay("MyModule.relay1")
relay = YRelay.FindRelay("MyModule.MyFunction")
relay = YRelay.FindRelay("MyFunction")
```

`YRelay.FindRelay` returns an object which you can then use at will to control the relay.

isOnline

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

---

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

set_state

The `set_state()` method of the objet returned by `yFindRelay` switches the relay position to one of its two outputs. The two possible parameter values are `Y_STATE_A` for output A, and `Y_STATE_B` for output B.

A real example

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

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

```vbnet
Module Module1

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

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

    If argv.Length < 3 Then Usage()

    target = argv(1)
    state = CChar(Mid(argv(2), 1, 1).ToUpper())

    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

    If target = "any" Then
      relay = YRelay.FirstRelay()
      If relay Is Nothing Then
        Console.WriteLine("No module connected (check USB cable) ")
        End
      End If
    Else
      relay = YRelay.FindRelay(target + ".relay1")
    End If

    If (relay.isOnline()) Then
      If state = "A" Then relay.set_state(Y_STATE_A) Else relay.set_state(Y_STATE_B)
    Else
      Console.WriteLine("Module not connected (check identification and USB cable)")
    End If
    YAPI.FreeAPI()
  End Sub

End Module
```

16.4. Control of the module part

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

```vbnet
Imports System.IO
```

```vb.net
Imports System.Environment

Module Module1

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

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

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

    If argv.Length < 2 Then usage()

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

End Module
```

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

Changing the module settings

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

```vb.net
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 = ""
```

```vbnet
    Dim newname As String
    Dim m As YModule

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

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

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

  End Sub

End Module
```

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

Listing the modules

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

```vbnet
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
```

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-LatchingRelay-C with Delphi or Lazarus

Delphi is a descendent of Turbo-Pascal. Originally, Delphi was produced by Borland, Embarcadero now edits it. The strength of this language resides in its ease of use, as anyone with some notions of the Pascal language can develop a Windows application in next to no time. Its only disadvantage is to cost something[1].

Lazarus[2] is a free IDE based on Free-Pascal, it has nothing to envy to Delphi and is available for both Windows and Linux. The Yoctopuce Delphi library is compatible with both Windows and Linux versions of Lazarus

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

17.1. Preparation

Go to the Yoctopuce web site and download the Yoctopuce Delphi libraries[4]. Uncompress everything in a directory of your choice.

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

Windows

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

---

[1] Actually, Borland provided free versions (for personal use) of Delphi 2006 and 2007. Look for them on the Internet, you may still be able to download them.
[2] www.lazarus-ide.org
[3] Delphi libraries are regularly tested with Delphi 5, Delphi XE2, and the latest version of Lazarus.
[4] www.yoctopuce.com/EN/libraries.php
[5] Use the **Tools / Environment options** menu.
[6] Use the Menu **Project / Project options/ Compiler options / Paths**

Linux

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

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

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

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

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

17.2. About examples

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

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

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

17.3. Control of the Relay function

A few lines of code are enough to use a Yocto-LatchingRelay-C. Here is the skeleton of a Delphi code snipplet to use the Relay function.

```delphi
uses yocto_api, yocto_relay;

var errmsg: string;
    relay: TYRelay;

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

// Retrieve the object used to interact with the device
relay = yFindRelay("LTCHRL2C-123456.relay1")

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

Let's look at these lines in more details.

yocto_api and yocto_relay

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

---

[7] See https://www.yoctopuce.com/EN/article/about-programming-examples

yRegisterHub

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

yFindRelay

The `yFindRelay` function allows you to find a relay 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-LatchingRelay-C module with serial number *LTCHRL2C-123456* which you have named "*MyModule*", and for which you have given the *relay1* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
relay := yFindRelay("LTCHRL2C-123456.relay1");
relay := yFindRelay("LTCHRL2C-123456.MyFunction");
relay := yFindRelay("MyModule.relay1");
relay := yFindRelay("MyModule.MyFunction");
relay := yFindRelay("MyFunction");
```

`yFindRelay` returns an object which you can then use at will to control the relay.

isOnline

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

set_state

The `set_state()` method of the objet returned by `yFindRelay` switches the relay position to one of its two outputs. The two possible parameter values are `Y_STATE_A` for output A, and `Y_STATE_B` for output B.

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;
{$APPTYPE CONSOLE}
uses
  SysUtils,
  {$IFNDEF UNIX}
  windows,
  {$ENDIF UNIX}
  yocto_api,
  yocto_relay;

Procedure  Usage();
  var
   exe : string;

  begin
    exe:= ExtractFileName(paramstr(0));
    WriteLn(exe+' <serial_number>  A|B');
    WriteLn(exe+' <logical_name> A|B');
    WriteLn(exe+' any A|B');
    WriteLn('');
    WriteLn('Example:');
    WriteLn(exe+' any B');
    sleep(3000);
    halt;
  End;
```

```
procedure setRelayState(relay:TYRelay; state:boolean);
  begin
    if (relay.isOnline()) then
     begin
       if state then relay.set_state(Y_STATE_B)
               else relay.set_state(Y_STATE_A);
     end
    else Writeln('Module not connected (check identification and USB cable)');
  end;

var

  relay     : TYRelay;
  errmsg    : string;

begin

  if (paramcount<2) then usage();

  // Setup the API to use local USB devices
  if yRegisterHub('usb', errmsg)<>YAPI_SUCCESS then
  begin
    Write('RegisterHub error: '+errmsg);
    sleep(3000);
    exit;
  end;

  if paramstr(1)='any' then
    begin
      // try to first the first relay available
      relay := yFirstRelay();
      if relay=nil then
        begin
          writeln('No module connected (check USB cable)');
          sleep(3000);
          halt;
        end
      end
  else // or use the one specified the command line
  relay:= YFindRelay(paramstr(1)+'.relay1');

  // make sure it connected
  if not relay.isOnline() then
    begin
       writeln('No module connected (check USB cable)');
       sleep(3000);
       halt;
    end;

   // lets drive the relay
  if paramstr(2)='B' then   setRelayState(relay,true)
                    else   setRelayState(relay,false);
  yFreeAPI();

end.
```

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.

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

const
  serial = 'LTCHRL2C-123456'; // use serial number or logical name

procedure refresh(module:Tymodule) ;
  begin
    if (module.isOnline())  then
     begin
```

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


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

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

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

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

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

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

Changing the module settings

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

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

const
  serial = 'LTCHRL2C-123456'; // use serial number or logical name

var
```

```
  module  : TYModule;
  errmsg  : string;
  newname : string;

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

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

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

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

Listing the modules

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

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

var
  module : TYModule;
  errmsg : string;

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

  Writeln('Device list');

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

```
end.
```

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 the Yocto-LatchingRelay-C with Universal Windows Platform

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

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

18.1. 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:

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

    return result;
}

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

---

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

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

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

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

18.2. Installation

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

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

```
<DeviceCapability Name="humaninterfacedevice">
    <!-- Yocto-LatchingRelay-C -->
    <Device Id="vidpid:24e0 00D9">
      <Function Type="usage:ff00 0001" />
```

---

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

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

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

18.4. Control of the Relay function

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

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

// Retrieve the object used to interact with the device
YRelay relay = YRelay.FindRelay("LTCHRL2C-123456.relay1");

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

[...]
```

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.

YRelay.FindRelay

The `YRelay.FindRelay` function allows you to find a relay 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-LatchingRelay-C module with serial number *LTCHRL2C-123456* which you have named "*MyModule*", and for which you have given the *relay1* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
relay = YRelay.FindRelay("LTCHRL2C-123456.relay1");
relay = YRelay.FindRelay("LTCHRL2C-123456.MaFonction");
relay = YRelay.FindRelay("MonModule.relay1");
relay = YRelay.FindRelay("MonModule.MaFonction");
relay = YRelay.FindRelay("MaFonction");
```

`YRelay.FindRelay` returns an object which you can then use at will to control the relay.

isOnline

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

set_state

The `set_state()` method of the objet returned by `YRelay.FindRelay` switches the relay position to one of its two outputs. The two possible parameter values are `YRelay.STATE_A` for output A, and `YRelay.STATE_B` for output B.

18.5. A real example

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

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

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

```csharp
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 RequestedState { get; set; }

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

        YRelay relay;

        if (Target.ToLower() == "any") {
          relay = YRelay.FirstRelay();
          if (relay == null) {
            WriteLine("No module connected (check USB cable) ");
            return -1;
          }
        } else {
          relay = YRelay.FindRelay(Target + ".relay1");
        }

        if (await relay.isOnline()) {
          if (RequestedState.ToUpper() == "A") {
            await relay.set_state(YRelay.STATE_A);
          } else {
            await relay.set_state(YRelay.STATE_B);
          }
        } else {
          WriteLine("Module not connected (check identification and USB cable)");
        }
      } catch (YAPI_Exception ex) {
        WriteLine("error: " + ex.Message);
      }

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

18.6. Control of the module part

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

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

namespace Demo
{
  public class Demo : DemoBase
  {

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

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

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

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

Each property `xxx` of the module can be read thanks to a method of type `YModule.get_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);
    }
    await YAPI.FreeAPI();
    return 0;
  }
}
}
```

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

Listing the modules

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

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

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

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

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

```
      } catch (YAPI_Exception ex) {
        WriteLine("Error:" + ex.Message);
      }
      await YAPI.FreeAPI();
      return 0;
    }
  }
}
```

18.7. 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);
}
```

19. Using Yocto-LatchingRelay-C with Objective-C

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

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

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

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

19.1. Control of the Relay function

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

```
#import "yocto_api.h"
#import "yocto_relay.h"

...
NSError *error;
[YAPI RegisterHub:@"usb": &error]
...
// On récupère l'objet représentant le module (ici connecté en local sur USB)
relay = [YRelay FindRelay:@"LTCHRL2C-123456.relay1"];

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

---

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

Let's look at these lines in more details.

yocto_api.h and yocto_relay.h

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

[YAPI RegisterHub]

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

[Relay FindRelay]

The `[Relay FindRelay]` function allows you to find a relay 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-LatchingRelay-C module with serial number *LTCHRL2C-123456* which you have named "*MyModule*", and for which you have given the *relay1* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
YRelay *relay = [Relay FindRelay:@"LTCHRL2C-123456.relay1"];
YRelay *relay = [Relay FindRelay:@"LTCHRL2C-123456.MyFunction"];
YRelay *relay = [Relay FindRelay:@"MyModule.relay1"];
YRelay *relay = [Relay FindRelay:@"MyModule.MyFunction"];
YRelay *relay = [Relay FindRelay:@"MyFunction"];
```

`[Relay FindRelay]` returns an object which you can then use at will to control the relay.

isOnline

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

set_state

The `set_state()` method of the objet returned by `YRelay.FindRelay` switches the relay position to one of its two outputs. The two possible parameter values are `YRelay.STATE_A` for output A, and `YRelay.STATE_B` for output B.

A real example

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

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

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

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

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

  if (argc < 3) {
```

```objc
    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]];
    NSString *state  = [NSString stringWithUTF8String:argv[2]];
    YRelay   *relay;

    if ([target isEqualToString:@"any"]) {
      relay =  [YRelay FirstRelay];
      if (relay == NULL) {
        NSLog(@"No module connected (check USB cable)");
        return 1;
      }
    } else {
      relay =  [YRelay FindRelay:target];
    }
    if ([relay isOnline]) {
      if ([state isEqualToString:@"A"])
        [relay set_state:Y_STATE_A];
      else
        [relay set_state:Y_STATE_B];
    } else {
      NSLog(@"Module not connected (check identification and USB cable)\n");
    }
    [YAPI FreeAPI];
  }
  return 0;
}
```

19.2. 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.

```objc
#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.

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

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

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

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

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

19.3. 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.

20. Using with unsupported languages

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

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

20.1. Command line

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

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

20.2. .NET Assembly

A .NET Assembly enables you to share a set of pre-compiled classes to offer a service, by stating entry points which can be used by third-party applications. In our case, it's the whole Yoctopuce library which is available in the .NET Assembly, so that it can be used in any environment which supports .NET Assembly dynamic loading.

The Yoctopuce library as a .NET Assembly does not contain only the standard C# Yoctopuce library, as this would not have allowed an optimal use in all environments. Indeed, we cannot expect host applications to necessarily offer a thread system or a callback system, although they are very useful to manage plug-and-play events and sensors with a high refresh rate. Likewise, we cannot expect from external applications a transparent behavior in cases where a function call in Assembly creates a delay because of network communications.

Therefore, we added to it an additional layer, called *.NET Proxy* library. This additional layer offers an interface very similar to the standard library but somewhat simplified, as it internally manages all the callback mechanisms. Instead, this library offers mirror objects, called *Proxys*, which publish through *Properties* the main attributes of the Yoctopuce functions such as the current measure, configuration parameters, the state, and so on.

.NET Assembly Architecture
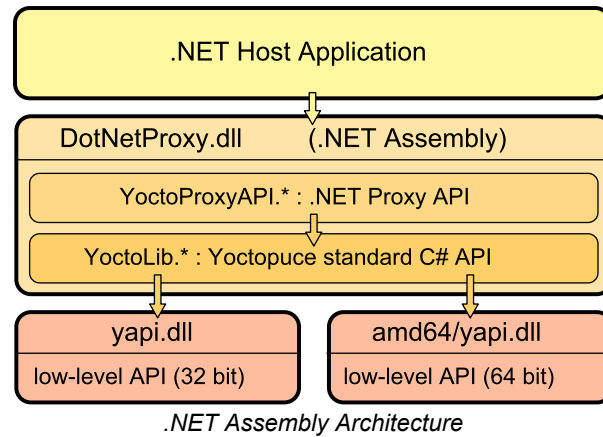
The callback mechanism automatically updates the properties of the *Proxys* objects, without the host application needing to care for it. The later can thus, at any time and without any risk of latency, display the value of all properties of Yoctopuce Proxy objects.

Pay attention to the fact that the `yapi.dll` low-level communication library is **not** included in the .NET Assembly. You must therefore keep it together with `DotNetProxyLibrary.dll`. The 32 bit version must be located in the same directory as `DotNetProxyLibrary.dll`, while the 64 bit version must be in a subdirectory `amd64`.

Example of use with MATLAB

Here is how to load our Proxy .NET Assembly in MATLAB and how to read the value of the first sensor connected by USB found on the machine:

```
NET.addAssembly("C:/Yoctopuce/DotNetProxyLibrary.dll");
import YoctoProxyAPI.*

errmsg = YAPIProxy.RegisterHub("usb");
sensor = YSensorProxy.FindSensor("");
measure = sprintf('%.3f %s', sensor.CurrentValue, sensor.Unit);
```

Example of use in PowerShell

PowerShell commands are a little stranger, but we can recognize the same structure:

```
Add-Type -Path "C:/Yoctopuce/DotNetProxyLibrary.dll"

$errmsg = [YoctoProxyAPI.YAPIProxy]::RegisterHub("usb")
$sensor = [YoctoProxyAPI.YSensorProxy]::FindSensor("")
$measure = "{0:n3} {1}" -f $sensor.CurrentValue, $sensor.Unit
```

Specificities of the .NET Proxy library

With regards to classic Yoctopuce libraries, the following differences in particular should be noted:

No FirstModule/nextModule method
To obtain an object referring to the first found module, we call `YModuleProxy.FindModule ("")`. If there is no connected module, this method returns an object with its `module.IsOnline` property set to `False`. As soon as a module is connected, the property changes to `True` and the module hardware identifier is updated.

To list modules, you can call the `module.GetSimilarFunctions()` method which returns an array of character strings containing the identifiers of all the modules which were found.

No callback function
Callback functions are implemented internally and they update the object properties. You can therefore simply poll the properties, without significant performance penalties. Be aware that if you

use one of the function that disables callbacks, the automatic refresh of object properties may not work anymore.

A new method `YAPIProxy.GetLog` makes it possible to retrieve low-level debug logs without using callbacks.

Enumerated types

In order to maximize compatibility with host applications, the .NET Proxy library does not use true .NET enumerated types, but simple integers. For each enumerated type, the library includes public constants named according to the possible values. Contrarily to standard Yoctopuce libraries, numeric values always start from `1`, as the value `0` is reserved to return an invalid value, for instance when the device is disconnected.

Invalid numeric results

For all numeric results, rather than using an arbitrary constant, the invalid value returned in case of error is *NaN*. You should therefore use function *isNaN()* to detect this value.

Using .NET Assembly without the Proxy library

If for a reason or another you do not want to use the Proxy library, and if your environment allows it, you can use the standard C# API as it is located in the Assembly, under the `YoctoLib` namespace. Beware however not to mix both types of use: either you go through the Proxy library, or you use he `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.

20.3. VirtualHub and HTTP GET

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

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

REST interface

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

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

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

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

Driving a module through the REST interface

Each Yoctopuce module has its own REST interface, available in several variants. Let us imagine a Yocto-LatchingRelay-C with the *LTCHRL2C-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/LTCHRL2C-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/LTCHRL2C-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/LTCHRL2C-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 relay1 function, build the following URL:

```
http://127.0.0.1:4444/bySerial/LTCHRL2C-12345/api/relay1.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/LTCHRL2C-12345/api/relay1/logicalName
```

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

```
http://127.0.0.1:4444/bySerial/LTCHRL2C-12345/api/relay1?logicalName=myFunction
```

You can find the list of available attributes for your Yocto-LatchingRelay-C at the beginning of the *Programming* chapter.

Accessing Yoctopuce data logger through the REST interface

This section only applies to devices with a built-in data logger.

The preview of all recorded data streams can be retrieved in JSON format using the following URL:

```
http://127.0.0.1:4444/bySerial/LTCHRL2C-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/LTCHRL2C-12345/dataLogger.json?id=relay1&utc=1389801080
```

20.4. Using dynamic libraries

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

Filename	Platform
libyapi.dylib	Max OS X
libyapi-amd64.so	Linux Intel (64 bits)
libyapi-armel.so	Linux ARM EL (32 bits)
libyapi-armhf.so	Linux ARM HL (32 bits)
libyapi-aarch64.so	Linux ARM (64 bits)
libyapi-i386.so	Linux Intel (32 bits)
yapi64.dll	Windows (64 bits)
yapi.dll	Windows (32 bits)

These dynamic libraries contain all the functions necessary to completely rebuild the whole high level API in any language able to integrate these libraries. This chapter nevertheless restrains itself to describing basic use of the modules.

Driving a module

The three essential functions of the low level API are the following:

```
int yapiInitAPI(int connection_type, char *errmsg);
int yapiUpdateDeviceList(int forceupdate, char *errmsg);
int yapiHTTPRequest(char *device, char *request, char* buffer,int buffsize,int *fullsize,
char *errmsg);
```

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

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

Finally, the *yapiHTTPRequest* function sends HTTP requests to the module REST API. The *device* parameter contains the serial number or the logical name of the module which you want to reach. The *request* parameter contains the full HTTP request (including terminal line breaks). *buffer* points to a character buffer long enough to contain the answer. *buffsize* is the size of the buffer. *fullsize* is a pointer to an integer to which will be assigned the actual size of the answer. The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message. This pointer can also point to *null*. The function returns a negative integer in case of error, zero otherwise.

The format of the requests is the same as the one described in the *VirtualHub et HTTP GET* section. All the character strings used by the API are strings made of 8-bit characters: Unicode and UTF8 are not supported.

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

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

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

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

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

const
  serial      = 'LTCHRL2C-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.

```pascal
// device description structure
type yDeviceSt = packed record
   vendorid          : word;
   deviceid          : word;
   devrelease        : word;
   nbinbterfaces     : word;
   manufacturer      : array [0..19] of ansichar;
   productname       : array [0..27] of ansichar;
   serial            : array [0..19] of ansichar;
   logicalname       : array [0..19] of ansichar;
   firmware          : array [0..21] of ansichar;
   beacon            : byte;
 end;

// Dll function import
function   yapiInitAPI(mode:integer;
                        errmsg : pansichar):integer;cdecl;
                        external 'yapi.dll' name 'yapiInitAPI';

function   yapiUpdateDeviceList(force:integer;errmsg : pansichar):integer;cdecl;
                        external 'yapi.dll' name 'yapiUpdateDeviceList';

function   yapiGetAllDevices( buffer:pointer;
                              maxsize:integer;
                              var neededsize:integer;
                              errmsg : pansichar):integer; cdecl;
                              external 'yapi.dll' name 'yapiGetAllDevices';

function   apiGetDeviceInfo(d:integer; var infos:yDeviceSt;
                              errmsg : pansichar):integer;  cdecl;
                              external 'yapi.dll' name 'yapiGetDeviceInfo';


var
 errmsgBuffer  : array [0..256] of ansichar;
 dataBuffer    : array [0..127] of integer;    // max of 128 USB devices
 errmsg,data   : pansichar;
 neededsize,i  : integer;
 devinfos      : yDeviceSt;

begin
  errmsg  :=  @errmsgBuffer;

  // API  initialization
  if(yapiInitAPI(1,errmsg)<0) then
   begin
    writeln(errmsg);
    halt;
   end;
```

```
  // forces a device inventory
if( yapiUpdateDeviceList(1,errmsg)<0) then
  begin
   writeln(errmsg);
   halt;
  end;

 // loads all device handles into dataBuffer
if yapiGetAllDevices(@dataBuffer,sizeof(dataBuffer),neededsize,errmsg)<0 then
  begin
   writeln(errmsg);
   halt;
  end;

 // gets device info from each handle
for i:=0 to  neededsize div sizeof(integer)-1 do
 begin
    if (apiGetDeviceInfo(dataBuffer[i], devinfos, errmsg)<0) then
      begin
        writeln(errmsg);
        halt;
      end;
    writeln(pansichar(@devinfos.serial)+' ('+pansichar(@devinfos.productname)+')');
  end;

end.
```

VB6 and yapi.dll

Each entry point from the yapi.dll is duplicated. You will find one regular C-decl version and one Visual Basic 6 compatible version, prefixed with *vb6_*.

20.5. 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.

21. Advanced programming

The preceding chapters have introduced, in each available language, the basic programming functions which can be used with your Yocto-LatchingRelay-C module. This chapter presents in a more generic manner a more advanced use of your module. Examples are provided in the language which is the most popular among Yoctopuce customers, that is C#. Nevertheless, you can find complete examples illustrating the concepts presented here in the programming libraries of each language.

To remain as concise as possible, examples provided in this chapter do not perform any error handling. Do not copy them "as is" in a production application.

21.1. 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.

22. High-level API Reference

This chapter summarizes the high-level API functions to drive your Yocto-LatchingRelay-C. Syntax and exact type names may vary from one language to another, but, unless otherwise stated, all the functions are available in every language. For detailed information regarding the types of arguments and return values for a given language, refer to the definition file for this language (`yocto_api.*` as well as the other `yocto_*` files that define the function interfaces).

For languages which support exceptions, all of these functions throw exceptions in case of error by default, rather than returning the documented error value for each function. This is by design, to facilitate debugging. It is however possible to disable the use of exceptions using the `yDisableExceptions()` function, in case you prefer to work with functions that return error values.

This chapter does not repeat the programming concepts described earlier, in order to stay as concise as possible. In case of doubt, do not hesitate to go back to the chapter describing in details all configurable attributes.

22.1. Class YAPI

General functions

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

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

`java`	**import com.yoctopuce.YoctoAPI.YAPI;**
`dnp`	**import YoctoProxyAPI.YAPIProxy**
`cp`	**#include "yocto_api_proxy.h"**
`ml`	**import YoctoProxyAPI.YAPIProxy"**
`cpp`	**#include "yocto_api.h"**
`vb`	**yocto_api.vb**
`cs`	**yocto_api.cs**
`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';**
`tpy`	**from yoctolib.yocto_api import ***
`vi`	**YModule.vi**
`pas`	**uses yocto_api;**
`es`	**in HTML: <script src="../../lib/yocto_api.js"></script>** **in node.js: require('yoctolib-es2017/yocto_api.js');**

Global functions
YAPI.AddTrustedCertificates(certificate)
Adds a TLS/SSL certificate to the list of trusted certificates.
`cpp` `vb` `cs` `java` `py` `php` `ts` `tpy` `pas` `es`
YAPI.AddUdevRule(force)
Adds a UDEV rule which authorizes all users to access Yoctopuce modules connected to the USB ports.
`cpp` `vb` `cs` `java` `py` `php` `ts` `pas` `es`
YAPI.CheckLogicalName(name)
Checks if a given string is valid as logical name for a module or a function.
`cpp` `vb` `cs` `java` `py` `php` `ts` `tpy` `pas` `es`
YAPI.ClearHTTPCallbackCacheDir(removeFiles)
Disables the HTTP callback cache.
`java` `php`
YAPI.DisableExceptions()
Disables the use of exceptions to report runtime errors.
`cpp` `vb` `cs` `py` `php` `ts` `tpy` `pas` `es`
YAPI.DownloadHostCertificate(url, mstimeout)
Download the TLS/SSL certificate from the hub.
`cpp` `vb` `cs` `java` `py` `php` `ts` `tpy` `pas` `es`

YAPI.EnableExceptions()

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

`cpp` `vb` `cs` `py` `php` `ts` `tpy` `pas` `es`

YAPI.EnableUSBHost(osContext)

This function is used only on Android.

`java`

YAPI.FreeAPI()

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

`cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `tpy` `pas` `es`

YAPI.GetAPIVersion()

Returns the version identifier for the Yoctopuce library in use.

`cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `tpy` `pas` `es`

YAPI.GetCacheValidity()

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

`cpp` `vb` `cs` `java` `py` `php` `ts` `tpy` `pas` `es`

YAPI.GetDeviceListValidity()

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

`cpp` `vb` `cs` `java` `py` `php` `ts` `tpy` `pas` `es`

YAPI.GetDllArchitecture()

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

`dnp`

YAPI.GetDllPath()

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

`dnp`

YAPI.GetLog(lastLogLine)

Retrieves Yoctopuce low-level library diagnostic logs.

`dnp`

YAPI.GetNetworkTimeout()

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

`cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `tpy` `pas` `es`

YAPI.GetTickCount()

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

`cpp` `vb` `cs` `java` `py` `php` `ts` `tpy` `pas` `es`

YAPI.GetYAPISharedLibraryPath()

Returns the path to the dynamic YAPI library.

`cpp` `vb` `cs` `java` `py` `php` `ts` `pas` `es`

YAPI.HandleEvents(errmsg)

Maintains the device-to-library communication channel.

`cpp` `vb` `cs` `java` `py` `php` `ts` `tpy` `pas` `es`

YAPI.InitAPI(mode, errmsg)

Initializes the Yoctopuce programming library explicitly.

cpp	vb	cs	java	py	php	ts	tpy	pas	es

YAPI.PreregisterHub(url, errmsg)

Fault-tolerant alternative to `yRegisterHub()`.

cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

YAPI.RegisterDeviceArrivalCallback(arrivalCallback)

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

cpp	vb	cs	java	py	php	ts	tpy	pas	es

YAPI.RegisterDeviceRemovalCallback(removalCallback)

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

cpp	vb	cs	java	py	php	ts	tpy	pas	es

YAPI.RegisterHub(url, errmsg)

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

cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

YAPI.RegisterHubDiscoveryCallback(hubDiscoveryCallback)

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

cpp	vb	cs	java	py	ts	tpy	pas	es

YAPI.RegisterHubWebsocketCallback(ws, errmsg, authpwd)

Variant to `yRegisterHub()` used to initialize Yoctopuce API on an existing Websocket session, as happens for incoming WebSocket callbacks.

YAPI.RegisterLogFunction(logfun)

Registers a log callback function.

cpp	vb	cs	java	py	ts	tpy	pas	es

YAPI.SelectArchitecture(arch)

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

py

YAPI.SetCacheValidity(cacheValidityMs)

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

cpp	vb	cs	java	py	php	ts	tpy	pas	es

YAPI.SetDelegate(object)

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

YAPI.SetDeviceListValidity(deviceListValidity)

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

cpp	vb	cs	java	py	php	ts	tpy	pas	es

YAPI.SetHTTPCallbackCacheDir(directory)

Enables the HTTP callback cache.

java	php

YAPI.SetNetworkSecurityOptions(opts)

Enables or disables certain TLS/SSL certificate checks.

cpp	vb	cs	java	py	php	ts	tpy	pas	es

YAPI.SetNetworkTimeout(networkMsTimeout)

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

`cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `tpy` `pas` `es`

YAPI.SetTimeout(callback, ms_timeout, args)

Invoke the specified callback function after a given timeout.

`ts` `es`

YAPI.SetTrustedCertificatesList(certificatePath)

Set the path of Certificate Authority file on local filesystem.

`cpp` `vb` `cs` `java` `py` `php` `ts` `tpy` `pas` `es`

YAPI.SetUSBPacketAckMs(pktAckDelay)

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

`java`

YAPI.Sleep(ms_duration, errmsg)

Pauses the execution flow for a specified duration.

`cpp` `vb` `cs` `java` `py` `php` `ts` `tpy` `pas` `es`

YAPI.TestHub(url, mstimeout, errmsg)

Test if the hub is reachable.

`cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `tpy` `pas` `es`

YAPI.TriggerHubDiscovery(errmsg)

Force a hub discovery, if a callback as been registered with `yRegisterHubDiscoveryCallback` it will be called for each net work hub that will respond to the discovery.

`cpp` `vb` `cs` `java` `py` `ts` `tpy` `pas` `es`

YAPI.UnregisterHub(url)

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

`cpp` `vb` `cs` `java` `py` `php` `ts` `tpy` `pas` `es`

YAPI.UpdateDeviceList(errmsg)

Triggers a (re)detection of connected Yoctopuce modules.

`cpp` `vb` `cs` `java` `py` `php` `ts` `tpy` `pas` `es`

YAPI.UpdateDeviceList_async(callback, context)

Triggers a (re)detection of connected Yoctopuce modules.

22.2. Class YModule

Global parameters control interface for all Yoctopuce devices

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

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

cpp	**#include "yocto_api.h"**
vb	**yocto_api.vb**
cs	**yocto_api.cs**
java	**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';**
dnp	**import YoctoProxyAPI.YModuleProxy**
cp	**#include "yocto_module_proxy.h"**
tpy	**from yoctolib.yocto_api import ***
vi	**YModule.vi**
ml	**import YoctoProxyAPI.YModuleProxy"**
pas	**uses yocto_api;**
es	**in HTML: <script src="../../lib/yocto_api.js"></script>** **in node.js: require('yoctolib-es2017/yocto_api.js');**

Global functions
YModule.FindModule(func)
Allows you to find a module from its serial number or from its logical name.
`cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `tpy` `pas` `es`
YModule.FindModuleInContext(yctx, func)
Retrieves a module for a given identifier in a YAPI context.
`java` `ts` `tpy` `es`
YModule.FirstModule()
Starts the enumeration of modules currently accessible.
`cpp` `vb` `cs` `java` `py` `php` `ts` `tpy` `pas` `es`
YModule properties
module→Beacon *[writable]*
State of the localization beacon.
`dnp`
module→FirmwareRelease *[read-only]*
Version of the firmware embedded in the module.
`dnp`
module→FunctionId *[read-only]*
Retrieves the hardware identifier of the *n*th function on the module.
`dnp`
module→HardwareId *[read-only]*

Unique hardware identifier of the module.

`dnp`

module→IsOnline *[read-only]*

Checks if the module is currently reachable.

`dnp`

module→LogicalName *[writable]*

Logical name of the module.

`dnp`

module→Luminosity *[writable]*

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

`dnp`

module→ProductId *[read-only]*

USB device identifier of the module.

`dnp`

module→ProductName *[read-only]*

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

`dnp`

module→ProductRelease *[read-only]*

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

`dnp`

module→SerialNumber *[read-only]*

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

`dnp`

`YModule` methods

module→addFileToHTTPCallback(filename)

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

`cmd` `cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `pas` `es`

module→checkFirmware(path, onlynew)

Tests whether the byn file is valid for this module.

`cmd` `cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `tpy` `pas` `es`

module→clearCache()

Invalidates the cache.

`cpp` `vb` `cs` `java` `py` `php` `ts` `tpy` `pas` `es`

module→describe()

Returns a descriptive text that identifies the module.

`cpp` `vb` `cs` `java` `py` `php` `ts` `pas` `es`

module→download(pathname)

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

`cmd` `cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `tpy` `pas` `es`

module→functionBaseType(functionIndex)

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

cpp	vb	cs	java	py	php	ts	tpy	pas	es

module→functionCount()

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

cpp	vb	cs	java	py	php	ts	tpy	pas	es

module→functionId(functionIndex)

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

cpp	vb	cs	java	py	php	ts	tpy	pas	es

module→functionName(functionIndex)

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

cpp	vb	cs	java	py	php	ts	tpy	pas	es

module→functionType(functionIndex)

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

cpp	vb	cs	java	py	php	ts	tpy	pas	es

module→functionValue(functionIndex)

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

cpp	vb	cs	java	py	php	ts	tpy	pas	es

module→get_allSettings()

Returns all the settings and uploaded files of the module.

cmd	cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→get_beacon()

Returns the state of the localization beacon.

cmd	cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→get_errorMessage()

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

cpp	vb	cs	java	py	php	ts	tpy	pas	es

module→get_errorType()

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

cpp	vb	cs	java	py	php	ts	tpy	pas	es

module→get_firmwareRelease()

Returns the version of the firmware embedded in the module.

cmd	cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→get_functionIds(funType)

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

cmd	cpp	vb	cs	java	py	php	ts	dnp	pas	es

module→get_hardwareId()

Returns the unique hardware identifier of the module.

cpp	vb	cs	java	py	php	ts	dnp	tpy	es	cmd	pas

module→get_icon2d()

Returns the icon of the module.

cmd	cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→get_lastLogs()

Returns a string with last logs of the module.

`cmd` `cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `tpy` `pas` `es`

module→get_logicalName()

Returns the logical name of the module.

`cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `tpy` `pas` `es` `cmd`

module→get_luminosity()

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

`cmd` `cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `tpy` `pas` `es`

module→get_parentHub()

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

`cmd` `cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `tpy` `pas` `es`

module→get_persistentSettings()

Returns the current state of persistent module settings.

`cmd` `cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `tpy` `pas` `es`

module→get_productId()

Returns the USB device identifier of the module.

`cmd` `cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `tpy` `pas` `es`

module→get_productName()

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

`cmd` `cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `tpy` `pas` `es`

module→get_productRelease()

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

`cmd` `cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `tpy` `pas` `es`

module→get_rebootCountdown()

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

`cmd` `cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `tpy` `pas` `es`

module→get_serialNumber()

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

`cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `tpy` `pas` `es` `cmd`

module→get_subDevices()

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

`cmd` `cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `pas` `es`

module→get_upTime()

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

`cmd` `cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `tpy` `pas` `es`

module→get_url()

Returns the URL used to access the module.

`cmd` `cpp` `vb` `cs` `java` `py` `php` `ts` `dnp` `tpy` `pas` `es`

module→get_usbCurrent()

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

cmd	cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→get_userData()

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

cpp	vb	cs	java	py	php	ts	tpy	pas	es

module→get_userVar()

Returns the value previously stored in this attribute.

cmd	cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→hasFunction(funcId)

Tests if the device includes a specific function.

cmd	cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→isOnline()

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

cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→isOnline_async(callback, context)

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

module→isReadOnly()

Indicates whether changes to the module are prohibited or allowed.

cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es	cmd

module→load(msValidity)

Preloads the module cache with a specified validity duration.

cpp	vb	cs	java	py	php	ts	tpy	pas	es

module→load_async(msValidity, callback, context)

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

module→log(text)

Adds a text message to the device logs.

cmd	cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→nextModule()

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

cpp	vb	cs	java	py	php	ts	tpy	pas	es

module→reboot(secBeforeReboot)

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

cmd	cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→registerBeaconCallback(callback)

Register a callback function, to be called when the localization beacon of the module has been changed.

cpp	vb	cs	java	py	php	ts	tpy	pas	es

module→registerConfigChangeCallback(callback)

Register a callback function, to be called when a persistent settings in a device configuration has been changed (e.g.

cpp	vb	cs	java	py	php	ts	tpy	pas	es

module→registerLogCallback(callback)

Registers a device log callback function.

cpp	vb	cs	java	py	php	ts	tpy	pas	es

module→revertFromFlash()

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

cmd	cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→saveToFlash()

Saves current settings in the nonvolatile memory of the module.

cmd	cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→set_allSettings(settings)

Restores all the settings of the device.

cmd	cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→set_allSettingsAndFiles(settings)

Restores all the settings and uploaded files to the module.

cmd	cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→set_beacon(newval)

Turns on or off the module localization beacon.

cmd	cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→set_logicalName(newval)

Changes the logical name of the module.

cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es	cmd

module→set_luminosity(newval)

Changes the luminosity of the module informative leds.

cmd	cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→set_userData(data)

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

cpp	vb	cs	java	py	php	ts	tpy	pas	es

module→set_userVar(newval)

Stores a 32 bit value in the device RAM.

cmd	cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→triggerConfigChangeCallback()

Triggers a configuration change callback, to check if they are supported or not.

cmd	cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→triggerFirmwareUpdate(secBeforeReboot)

Schedules a module reboot into special firmware update mode.

cmd	cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→updateFirmware(path)

Prepares a firmware update of the module.

cmd	cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→updateFirmwareEx(path, force)

Prepares a firmware update of the module.

cmd	cpp	vb	cs	java	py	php	ts	dnp	tpy	pas	es

module→wait_async(callback, context)

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

`ts` `es`

22.3. Class YRelay

Relay control interface, available for instance in the Yocto-LatchedRelay, the Yocto-MaxiPowerRelay, the Yocto-PowerRelay-V3 or the Yocto-Relay

The `YRelay` class allows you to drive a Yoctopuce relay or optocoupled output. It can be used to simply switch the output on or off, but also to automatically generate short pulses of determined duration. On devices with two output for each relay (double throw), the two outputs are named A and B, with output A corresponding to the idle position (normally closed) and the output B corresponding to the active state (normally open).

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

es	**in HTML: <script src="../../lib/yocto_relay.js"></script>** **in node.js: require('yoctolib-es2017/yocto_relay.js');**
cpp	**#include "yocto_relay.h"**
vb	**yocto_relay.vb**
cs	**yocto_relay.cs**
java	**import com.yoctopuce.YoctoAPI.YRelay;**
py	**from yocto_relay import ***
php	**require_once('yocto_relay.php');**
ts	**in HTML: import { YRelay } from '../../dist/esm/yocto_relay.js';** **in Node.js: import { YRelay } from 'yoctolib-cjs/yocto_relay.js';**
dnp	**import YoctoProxyAPI.YRelayProxy**
cp	**#include "yocto_relay_proxy.h"**
tpy	**from yoctolib.yocto_relay import ***
vi	**YRelay.vi**
ml	**import YoctoProxyAPI.YRelayProxy**
pas	**uses yocto_relay;**

Global functions

YRelay.FindRelay(func)

Retrieves a relay for a given identifier.

| cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es |

YRelay.FindRelayInContext(yctx, func)

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

| java | ts | tpy | es |

YRelay.FirstRelay()

Starts the enumeration of relays currently accessible.

| cpp | vb | cs | java | py | php | ts | tpy | pas | es |

YRelay.FirstRelayInContext(yctx)

Starts the enumeration of relays currently accessible.

| java | ts | tpy | es |

YRelay.GetSimilarFunctions()

Enumerates all functions of type Relay available on the devices currently reachable by the library, and returns their unique hardware ID.

| dnp |

YRelay properties

---

relay→AdvertisedValue *[read-only]*

Short string representing the current state of the function.

`dnp`

relay→FriendlyName *[read-only]*

Global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

`dnp`

relay→FunctionId *[read-only]*

Hardware identifier of the relay, without reference to the module.

`dnp`

relay→HardwareId *[read-only]*

Unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`.

`dnp`

relay→IsOnline *[read-only]*

Checks if the function is currently reachable.

`dnp`

relay→LogicalName *[writable]*

Logical name of the function.

`dnp`

relay→MaxTimeOnStateA *[writable]*

Maximum time (ms) allowed for the relay to stay in state A before automatically switching back in to B state.

`dnp`

relay→MaxTimeOnStateB *[writable]*

The maximum time (ms) allowed for the relay to stay in state B before automatically switching back in to A state.

`dnp`

relay→SerialNumber *[read-only]*

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

`dnp`

relay→State *[writable]*

State of the relays (A for the idle position, B for the active position).

`dnp`

relay→StateAtPowerOn *[writable]*

State of the relays at device startup (A for the idle position, B for the active position, UNCHANGED to leave the relay state as is).

`dnp`

YRelay methods

relay→clearCache()

Invalidates the cache.

`cpp` `vb` `cs` `java` `py` `php` `ts` `tpy` `pas` `es`

relay→delayedPulse(ms_delay, ms_duration)

Schedules a pulse.

| cmd | cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es |

relay→describe()

Returns a short text that describes unambiguously the instance of the relay in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

| cpp | vb | cs | java | py | php | ts | pas | es |

relay→get_advertisedValue()

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

| cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es | cmd |

relay→get_countdown()

Returns the number of milliseconds remaining before a pulse (delayedPulse() call) When there is no scheduled pulse, returns zero.

| cmd | cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es |

relay→get_errorMessage()

Returns the error message of the latest error with the relay.

| cpp | vb | cs | java | py | php | ts | tpy | pas | es |

relay→get_errorType()

Returns the numerical error code of the latest error with the relay.

| cpp | vb | cs | java | py | php | ts | tpy | pas | es |

relay→get_friendlyName()

Returns a global identifier of the relay in the format `MODULE_NAME.FUNCTION_NAME`.

| cpp | cs | java | py | php | ts | dnp | tpy | es |

relay→get_functionDescriptor()

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

| cpp | vb | cs | java | py | php | ts | pas | es |

relay→get_functionId()

Returns the hardware identifier of the relay, without reference to the module.

| cpp | vb | cs | java | py | php | ts | dnp | tpy | es |

relay→get_hardwareId()

Returns the unique hardware identifier of the relay in the form `SERIAL.FUNCTIONID`.

| cpp | vb | cs | java | py | php | ts | dnp | tpy | es |

relay→get_logicalName()

Returns the logical name of the relay.

| cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es | cmd |

relay→get_maxTimeOnStateA()

Returns the maximum time (ms) allowed for the relay to stay in state A before automatically switching back in to B state.

| cmd | cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es |

relay→get_maxTimeOnStateB()

Retourne the maximum time (ms) allowed for the relay to stay in state B before automatically switching back in to A state.

| cmd | cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es |

relay→get_module()

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

| cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es |

relay→get_module_async(callback, context)

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

relay→get_output()

Returns the output state of the relays, when used as a simple switch (single throw).

| cmd | cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es |

relay→get_pulseTimer()

Returns the number of milliseconds remaining before the relays is returned to idle position (state A), during a measured pulse generation.

| cmd | cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es |

relay→get_serialNumber()

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

| cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es | cmd |

relay→get_state()

Returns the state of the relays (A for the idle position, B for the active position).

| cmd | cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es |

relay→get_stateAtPowerOn()

Returns the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED to leave the relay state as is).

| cmd | cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es |

relay→get_userData()

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

| cpp | vb | cs | java | py | php | ts | tpy | pas | es |

relay→isOnline()

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

| cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es |

relay→isOnline_async(callback, context)

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

relay→isReadOnly()

Indicates whether changes to the function are prohibited or allowed.

| cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es | cmd |

relay→load(msValidity)

Preloads the relay cache with a specified validity duration.

| cpp | vb | cs | java | py | php | ts | tpy | pas | es |

relay→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

| cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es |

relay→load_async(msValidity, callback, context)

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

relay→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

| cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es | cmd |

relay→nextRelay()

Continues the enumeration of relays started using `yFirstRelay()`.

| cpp | vb | cs | java | py | php | ts | tpy | pas | es |

relay→pulse(ms_duration)

Sets the relay to output B (active) for a specified duration, then brings it automatically back to output A (idle state).

| cmd | cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es |

relay→registerValueCallback(callback)

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

| cpp | vb | cs | java | py | php | ts | tpy | pas | es |

relay→set_logicalName(newval)

Changes the logical name of the relay.

| cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es | cmd |

relay→set_maxTimeOnStateA(newval)

Changes the maximum time (ms) allowed for the relay to stay in state A before automatically switching back in to B state.

| cmd | cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es |

relay→set_maxTimeOnStateB(newval)

Changes the maximum time (ms) allowed for the relay to stay in state B before automatically switching back in to A state.

| cmd | cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es |

relay→set_output(newval)

Changes the output state of the relays, when used as a simple switch (single throw).

| cmd | cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es |

relay→set_state(newval)

Changes the state of the relays (A for the idle position, B for the active position).

| cmd | cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es |

relay→set_stateAtPowerOn(newval)

Changes the state of the relays at device startup (A for the idle position, B for the active position, UNCHANGED to leave the relay state as is).

| cmd | cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es |

relay→set_userData(data)

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

| cpp | vb | cs | java | py | php | ts | tpy | pas | es |

relay→toggle()

Switch the relay to the opposite state.

| cmd | cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es |

relay→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

| cpp | vb | cs | java | py | php | ts | dnp | tpy | pas | es | cmd |

relay→wait_async(callback, context)

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

ts | es

relay→wait_async(callback, context)

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

ts | es

23. Troubleshooting

23.1. 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].

23.2. 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].

23.3. Linux and USB

To work correctly under Linux, the library needs to have write access to all the Yoctopuce USB peripherals. However, by default under Linux, USB privileges of the non-root users are limited to read access. To avoid having to run the library as root, you need to create a new *udev* rule to authorize one or several users to have write access to the Yoctopuce peripherals.

To add a new *udev* rule to your installation, you must add a file with a name following the "`##-arbitraryName.rules`" format, in the "`/etc/udev/rules.d`" directory. When the system is starting, *udev* reads all the files with a "`.rules`" extension in this directory, respecting the alphabetical order (for example, the "`51-custom.rules`" file is interpreted AFTER the "`50-udev-default.rules`" file).

The "`50-udev-default`" file contains the system default *udev* rules. To modify the default behavior, you therefore need to create a file with a name that starts with a number larger than 50, that will override the system default rules. Note that to add a rule, you need a root access on the system.

In the `udev_conf` directory of the VirtualHub for Linux[3] archive, there are two rule examples which you can use as a basis.

---

[1] see: http://www.yoctopuce.com/EN/blog_by_categories/for-the-beginners
[2] see: http://www.yoctopuce.com/EN/article/about-programming-examples
[3] http://www.yoctopuce.com/FR/virtualhub.php

Example 1: 51-yoctopuce.rules

This rule provides all the users with read and write access to the Yoctopuce USB devices. Access rights for all other devices are not modified. If this scenario suits you, you only need to copy the "`51-yoctopuce_all.rules`" file into the "`/etc/udev/rules.d`" directory and to restart your system.

```
# udev rules to allow write access to all users
# for Yoctopuce USB devices
SUBSYSTEM=="usb", ATTR{idVendor}=="24e0", MODE="0666"
```

Example 2: 51-yoctopuce_group.rules

This rule authorizes the "yoctogroup" group to have read and write access to Yoctopuce USB peripherals. Access rights for all other peripherals are not modified. If this scenario suits you, you only need to copy the "`51-yoctopuce_group.rules`" file into the "`/etc/udev/rules.d`" directory and restart your system.

```
# udev rules to allow write access to all users of "yoctogroup"
# for Yoctopuce USB devices
SUBSYSTEM=="usb", ATTR{idVendor}=="24e0", MODE="0664", GROUP="yoctogroup"
```

23.4. ARM Platforms: HF and EL

There are two main flavors of executable on ARM: HF (Hard Float) binaries, and EL (EABI Little Endian) binaries. These two families are not compatible at all. The compatibility of a given ARM platform with of one of these two families depends on the hardware and on the OS build. ArmHL and ArmEL compatibility problems are quite difficult to detect. Most of the time, the OS itself is unable to make a difference between an HF and an EL executable and will return meaningless messages when you try to use the wrong type of binary.

All pre-compiled Yoctopuce binaries are provided in both formats, as two separate ArmHF et ArmEL executables. If you do not know what family your ARM platform belongs to, just try one executable from each family.

23.5. Powered module but invisible for the OS

If your Yocto-LatchingRelay-C is connected by USB, if its blue led is on, but if the operating system cannot see the module, check that you are using a true USB cable with data wires, and not a charging cable. Charging cables have only power wires.

23.6. Another process named xxx is already using yAPI

If when initializing the Yoctopuce API, you obtain the "*Another process named xxx is already using yAPI*" error message, it means that another application is already using Yoctopuce USB modules. On a single machine only one process can access Yoctopuce modules by USB at a time. You can easily work around this limitation by using VirtualHub and the network mode [4].

23.7. Disconnections, erratic behavior

If your Yocto-LatchingRelay-C behaves erratically and/or disconnects itself from the USB bus without apparent reason, check that it is correctly powered. Avoid cables with a length above 2 meters. If needed, insert a powered USB hub [5] [6].

---

[4] see: http://www.yoctopuce.com/EN/article/error-message-another-process-is-already-using-yapi
[5] see: http://www.yoctopuce.com/EN/article/usb-cables-size-matters
[6] see: http://www.yoctopuce.com/EN/article/how-many-usb-devices-can-you-connect

23.8. After a failed firmware update, the device stopped working

If a firmware update of your Yocto-LatchingRelay-C fails, it is possible that the module is no longer working. If this is the case, plug in your module while holding down the Yocto-Button. The Yocto-LED should light up brightly and remain steady. Release the button. Your Yocto-LatchingRelay-C should then appear at the bottom of the VirtualHub user interface as a module waiting to be flashed. This operation also reverts the module to its factory configuration.

23.9. The web interface shows errors after a firmware update

After an update, the windows corresponding to Yocto-LatchingRelay-C in the VirtualHub user interface report errors. This might be a bug, but it is more likely that your web browser has cached part of the interface code from the previous firmware. Perform a *shift-reload* or clear your browser cache and everything should be back to normal.

23.10. Registering VirtualHub disconnects another instance

If, when performing a call to RegisterHub() with a VirtualHub address, another previously registered VirtualHub disconnects, make sure the machine running these VirtualHubs do not have the same *Hostname*. Same *Hostname* can happen very easily when the operating system is installed from a monolithic image, Raspberry Pi are the best example. The Yoctopuce API uses serial numbers to communicate with devices and VirtualHub serial numbers are created on the fly based the *hostname* of the machine running VirtualHub.

23.11. Dropped commands

If, after sending a bunch of commands to a Yoctopuce device, you are under the impression that the last ones have been ignored, a typical example is a quick and dirty program meant to configure a device, make sure you used a YAPI.FreeAPI() at the end of the program. Commands are sent to Yoctopuce modules asynchronously thanks to a background thread. When the main program terminates, that thread is killed no matter if some command are left to be sent. However API.FreeAPI() waits until there is no more command to send before freeing the API resources and returning.

23.12. Damaged device

Yoctopuce strives to reduce the production of electronic waste. If you believe that your Yocto-LatchingRelay-C is not working anymore, start by contacting Yoctopuce support by e-mail to diagnose the failure. Even if you know that the device was damaged by mistake, Yoctopuce engineers might be able to repair it, and thus avoid creating electronic waste.

Waste Electrical and Electronic Equipment (WEEE) If you really want to get rid of your Yocto-LatchingRelay-C, do not throw it away in a trash bin but bring it to your local WEEE recycling point. In this way, it will be disposed properly by a specialized WEEE recycling center.
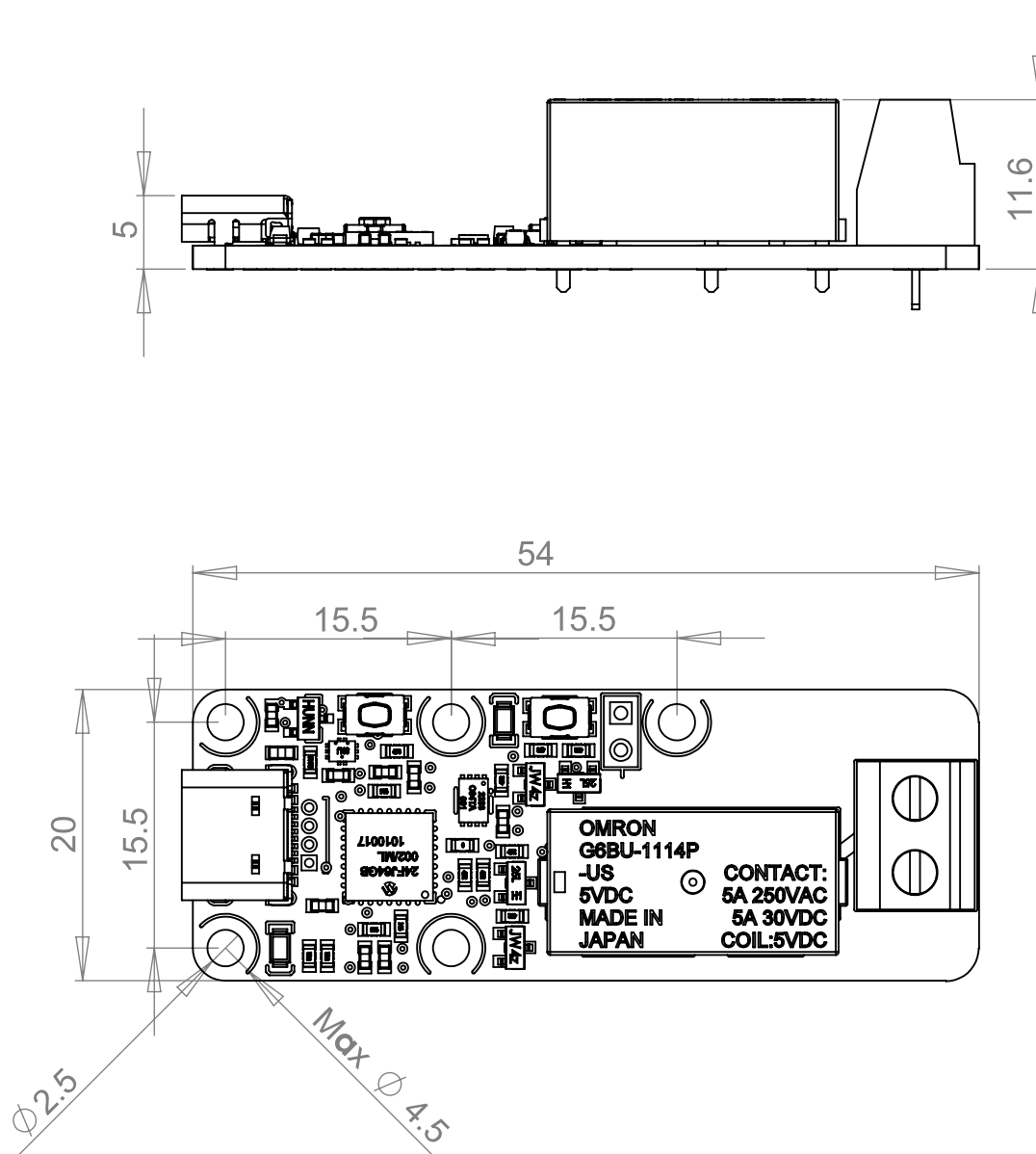
24. Characteristics

You can find below a summary of the main technical characteristics of your Yocto-LatchingRelay-C module.

Product ID	LTCHRL2C
Hardware release[†]	
USB connector	USB-C
Width	20 mm
Length	54 mm
Weight	9.7 g
Relay type	OMRON G6BU-1114P
Max switching current	5 A
Max switching power	1250 W
Max working voltage	250 V
Protection class, according to IEC 61140	Classe II
USB isolation, clearance dist.	4.1 mm
USB isolation, creepage dist.	5 mm
USB isolation, dielectric strength	3 kV
Normal operating temperature	5...40 °C
Extended operating temperature[‡]	-25..70 °C
RoHS compliance	RoHS III (2011/65/UE+2015/863)
USB Vendor ID	0x24E0
USB Device ID	0x00D9
Suggested enclosure	Yoctobox-long-thick-black
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.

5

11.6

54

15.5   15.5

20

15.5

Ø 2.5

Max Ø 4.5

OMRON
G6BU-1114P
-US
5VDC
MADE IN
JAPAN

CONTACT:
5A 250VAC
5A 30VDC
COIL:5VDC

All dimensions are in mm
Toutes les dimensions sont en mm

Yocto-LatchingRelay-C

A4

Scale
2:1
Echelle