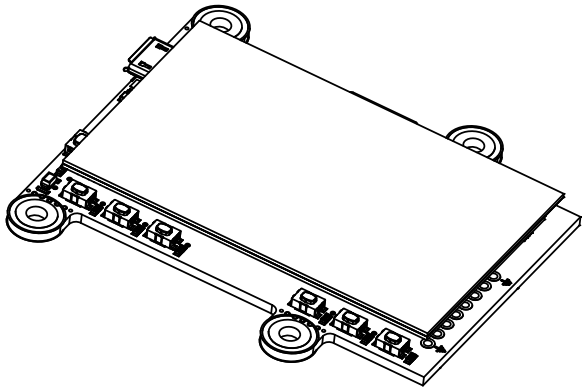
Yocto-MaxiDisplay-G

User manual

Table of contents

1. Introduction

The Yocto-MaxiDisplay-G is a 66x58mm electronic module allowing you to drive a 128x64px monochrome OLED screen, as well as 6 channels which can measure the state of switches, push buttons, or even potentiometers. This screen enables you to easily display some easily readable pieces of information from a machine which does not usually have a monitor.



The Yocto-MaxiDisplay-G module

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

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

1.1. Safety Information

The Yocto-MaxiDisplay-G 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

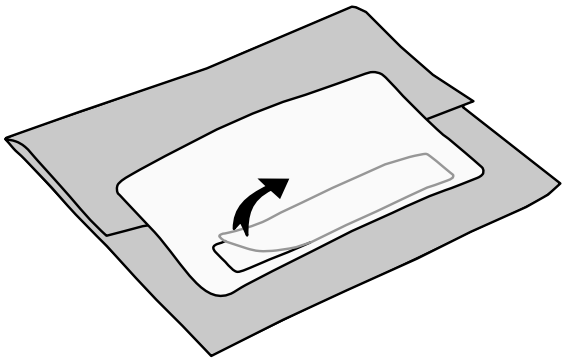
The Yocto-MaxiDisplay-G should not be used without a protective enclosure, because of the accessible bare electronic components. For optimal safety, it should be put into a non-metallic, non-inflammable enclosure, resistant to a mechanical stress level of 5 J. For instance, use a polycarbonate (e.g.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

Maintenance

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

Identification

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



Identification label is integrated in the package label.

Application

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

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

Environment

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

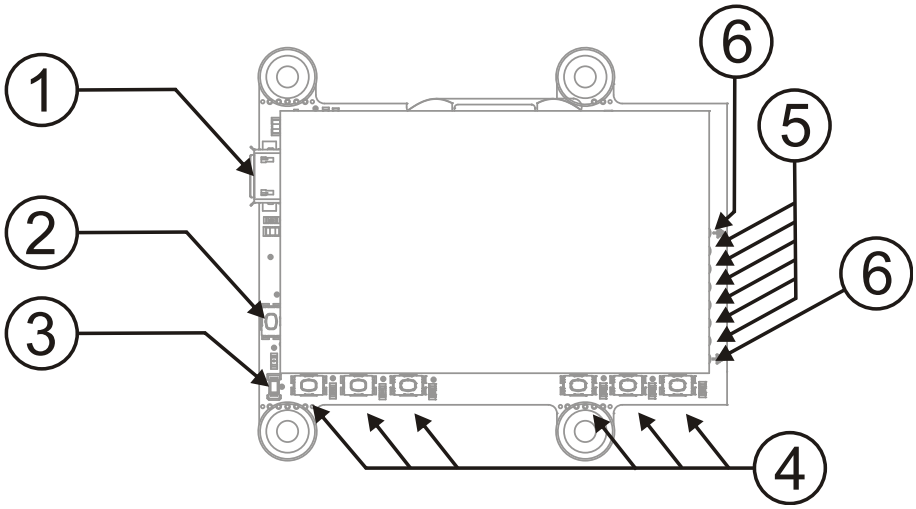
1.2. Environmental conditions

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

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

The extended operating temperature range for the Yocto-MaxiDisplay-G is -25...70°C. This temperature range has been determined based on components manufacturer recommendations, and on controlled environment tests performed during a limited duration (1h). If you plan to use the Yocto-MaxiDisplay-G in harsh environments for a long period of time, we strongly advise you to run extensive tests before going to production.

2. Presentation



1: Micro-B USB socket
2: Yocto-button
3: Yocto-led
4: Test anButtons 1 to 6
5: anButtons inputs 1 to 6
6: Common ground

2.1. Common elements

All Yocto-modules share a number of common functionalities.

USB connector

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

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

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

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

Yocto-button

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

Yocto-led

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

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

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

Current sensor

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

Serial number

Each Yocto-module has a unique serial number assigned to it at the factory. For Yocto-MaxiDisplay-G modules, this number starts with YD128G64. The module can be software driven using this serial number. The serial number cannot be modified.

Logical name

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

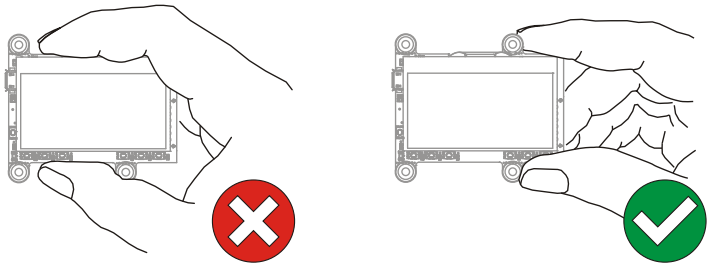
2.2. Specific elements

The screen

The screen is an OLED screen manufactured by WiseChip under the UG-2864ASGGG14 reference. Being made of glass, it is rather fragile. Do not let your Yocto-MaxiDisplay-G drop. When you install your Yocto-MaxiDisplay-G, make sure that the screen is not subjected to any mechanical constraint. Moreover, the ribbon coming out of the screen is particularly fragile at its junction with the screen.

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

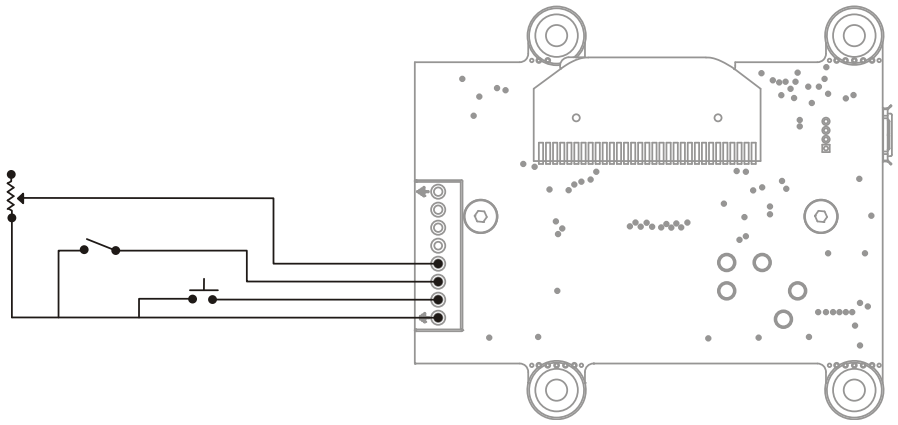
Make sure it is not subject to any mechanical constraint. When you manipulate the Yocto-MaxiDisplay-G, do not press on the ribbon.



Do not press on the ribbon when you manipulate the Yocto-MaxiDisplay-G.

Inputs 1 to 6

The Yocto-MaxiDisplay-G module contains 6 inputs allowing you to measure the state of resistive components (switches, push buttons, potentiometers, and so on). These inputs share a common ground. This means that each switch / push button / potentiometer must be connected both to the corresponding input and to the common ground. You can use any potentiometer value between 1KΩ and 200 KΩ.



Wiring of a potentiometer, a switch, and a push button, with a common ground.

You will probably want to solder a connector at the designed location. To access the pads, remove the screws located at the back of your Yocto-MaxiDisplay-G, delicately move the screen aside, and solder your connector. Then put the screen back in position.

Potentiometers and calibration
This module allows you to use a wide range of potentiometer values. But to enable it to provide you with coherent measures for the model you are using, you must calibrate the corresponding channels. You can do this very easily thanks to the configuration interface. You do not need to perform a calibration if you use only simple switches or push buttons.

Test push buttons
Each channel has its own small push button allowing you to artificially close the corresponding circuit. This is helpful to debug your projects.

2.3. Optional accessories

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

Screws and spacers

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

Micro-USB hub

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

YoctoHub-Ethernet, YoctoHub-Wireless and YoctoHub-GSM

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

1.27mm (or 1.25mm) connectors

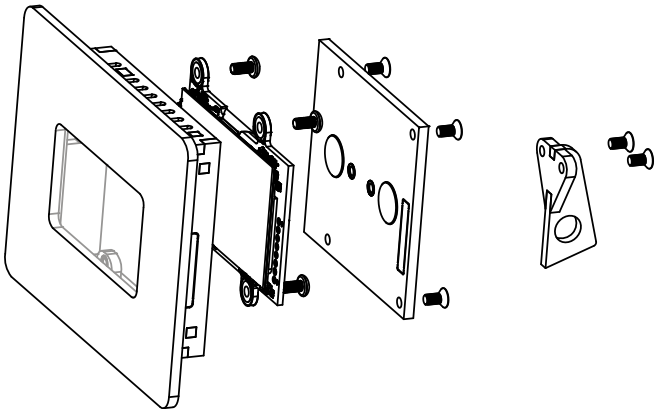
In case you wish to connect your Yocto-MaxiDisplay-G to a Micro-hub USB or a YoctoHub without using a bulky USB connector, you can use the four 1.27mm pads just behind the USB connector. There are two options.

You can mount the Yocto-MaxiDisplay-G 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-MaxiDisplay-G.

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-MaxiDisplay-G has been designed to be installed as is in your project. Nevertheless, Yoctopuce sells enclosures specifically designed for Yoctopuce modules. The recommended enclosure for your Yocto-MaxiDisplay-G is the YoctoBox-MaxiDisplay model. It has a small removable stand enabling it to stay upright, it is also equipped with powerful magnets allowing it to stick on ferromagnetic surfaces. More details are available on the Yoctopuce web site [3].



You can install your Yocto-MaxiDisplay-G in an optional enclosure.

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

3. Working principles

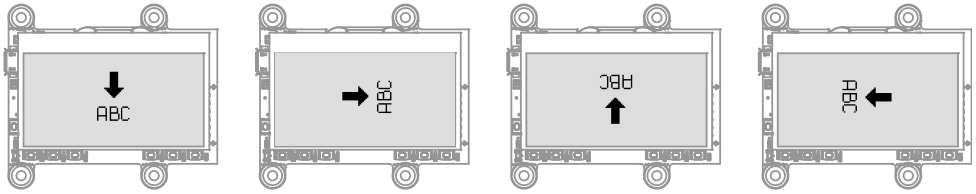
3.1. Embedded processor and memory

Like all the Yoctopuce modules, your Yocto-MaxiDisplay-G contains an embedded processor allowing it to perform relatively complex operations transparently. Thus, to draw a line, the host computer only needs to send a *draw a line* command to the Yocto-MaxiDisplay-G. It does not have to do anything else, everything is managed by the Yocto-MaxiDisplay-G processor. For this reason, the Yocto-MaxiDisplay-G behaves a little like a graphic accelerator where the graphical tasks are performed by a dedicated processor, letting the main processor perform other tasks.

Your Yocto-MaxiDisplay-G contains also a small file system to help you store some graphics, fonts, and other animations.

3.2. Orientation

To facilitate its hardware installation, your Yocto-MaxiDisplay-G can work in four distinct orientations. You only need to set a single parameter. When you set the value of this parameter to *left*, *up*, *right*, or *down* to indicate the position of the USB socket with regards to the circuit, the screen rotates its content in order for it to appear in the correct orientation.

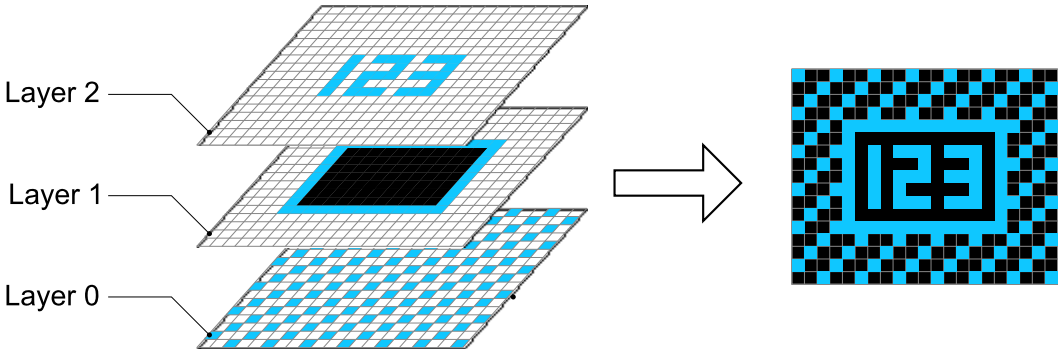


Parameter values LEFT, UP, RIGHT, and DOWN influence the orientation of the display.

This parameter is persistent and can be saved in the flash memory of the Yocto-MaxiDisplay-G.

3.3. Layer system

Your Yocto-MaxiDisplay-G works according to a principle of superposed and independent layers. You can write and draw independently in each of the 5 layers. This allows you to simplify and optimize your display code.



Several display layers are available.

You can hide or show any layer. You can even laterally move these layers which are slightly larger than the displayable surface (128x128), generating thus a scrolling effect. You can take advantage of this layer system to implement a *double buffering*[1] system.

Each layer has its own graphical context: cursor position, current font, current color, etc... This means that you must set these parameters for each layer with which you work. But this also means that several distinct processes can interact with your Yocto-MaxiDisplay-G without risking conflict issues: they only need to write in different layers.

Primitives working directly on display layers include:

- clear
- hide
- unhide
- setLayerPosition
- reset
- swapLayerContent
- copyLayerContent

3.4. Graphic routines

Your Yocto-MaxiDisplay-G contains basic graphic routines: lines, rectangles, circles, discs, text display, etc. All these routines support clipping: you can write on top of a layer border, the part located in the zone managed is taken into account, the outside part is ignored.

For more complex graphical operations, or simply if you feel more comfortable with it, you can also use your favorite graphic library running on the host driving the Yocto-MaxiDisplay-G to build a bitmap in memory, then render it with a single command on the layer of your choice. The Yoctopuce API is fast enough to make this possible even to make real-time animations.

Basic graphic primitives are:

- moveTo
- lineTo
- drawPixel
- drawRect
- drawBar
- drawCircle
- drawDisc
- drawBitmap
- drawImage

Colors

The Yocto-MaxiDisplay-G screen is purely monochrome. You cannot, therefore, display grey levels, nor benefit from anti-aliasing. You can draw using three "colors": the screen display color (which we

[1] http://en.wikipedia.org/wiki/Multiple_buffering

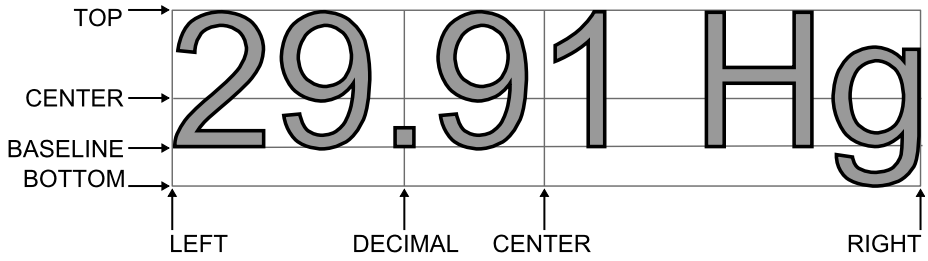
will call "white" in this documentation even if it might be light blue for example), black, or transparent (eraser).When you write using the transparent color, the layer below becomes visible. Note that layer 0 is not transparent. Writing in transparent color on this layer is equivalent to writing in black. This is important when you switch the content of two layers.

Primitives allowing you to change the color of a drawing are:

- selectGrayPen
- selectEraser

3.5. Text display

You can display any text at an arbitrary location of the screen. The Yocto-MaxiDisplay-G contains some embedded fonts, but you can create your own relatively easily. It is not possible to know beforehand the size of a text, but to compensate this, numerous text alignment modes are available. You can align text left, right, and center, from the decimal point, the base line, etc.



The alignment options

Primitives allowing you to display text are:

- selectFont
- drawText

Fonts built-in into the device firmware are:

- `Small.yfm` (height: 8 pixels)
- `Medium.yfm` (height: 16 pixels)
- `Large.yfm` (height: 32 pixels)
- `8x8.yfm` (monospaced)

Console mode

There is another method to display text on your Yocto-MaxiDisplay-G: the console mode. The console is a rectangular area of which you can parameterize the position. Texts displayed in this console are displayed like in a terminal, and line feeds are automatically generated. By default, the console size of each layer is initialized to the size of the screen.

Primitives allowing you to manage the console are:

- clearConsole
- consoleOut
- setConsoleMargins
- setConsoleBackground
- setConsoleWordWrap

Internationalization and regional characters

The text display functions can render international characters for languages that meet the following criteria:

- 8-bit character set

- left-to-right writing

For character sets requiring more than 8-bit (for instance chinese characters) and for right-to-left (RTL) languages, the only solution is to create a bitmap image on the computer using system built-in functions, and to display it using the `drawBitmap` primitive.

For all other languages, you should simply make sure that you use using the same locale (or *code page*) for the display font and for the text strings to draw. With this, you will be able to display all kind of accented characters and glyphs. Built-in fonts are provided according to the `iso-8859-1` locale (also known as `iso-latin-1` or `Windows-1252`), and support all languages from Western Europe. But you can easily generate equivalent fonts for the locale used by your computer using the small utility provided in the Delphi Library, under *Examples\Display-font-generator*.

In practice, for all languages with native Unicode support (e.g. 16-bit or UTF-8) like Python, C#, VB or Java, you can configure in the API the codepage to use to convert from unicode character strings to 8-bit format. The default value is `iso-8859-1`, corresponding to the built-in fonts.

For other languages like Delphi, C++ or PHP where the codepage is implicitly determined by the encoding of the source file, and where conversions are under the responsibility of the developer, you must simply take care to provide to the API a string that is compatible with the character set used in your display. The most common pitfall is to use an UTF-8 encoded string (because this is the most common format used by editors nowadays) and to forget to convert it into `iso-8859-*` before using it with `drawText` or `consoleOut`. If your display shows two strange letters instead of every accented character, this is the reason.

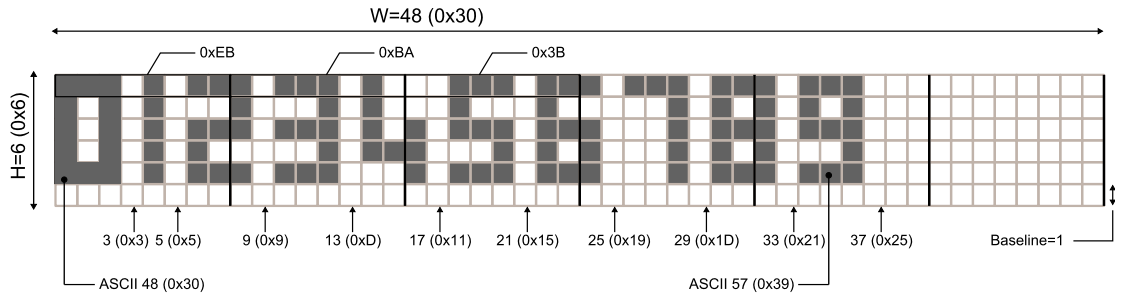
3.6. Font file format

Your Yocto-MaxiDisplay-G contains a few embedded fonts, but it is designed so that you could create your own fonts as easily as possible. A font file for your Yocto-MaxiDisplay-G is mostly a large bitmap where all the characters are drawn one after the other, in the order of the ASCII code of each character. Besides the bitmap, these files include a header with a some more information, as well as a list of the position of the last column of each character. The format is the following:

offset	type	Size (bytes)	signification
0x00	U16	2	Signature ("YF" 0x4659, little endian)
0x02	U8	1	Version = 1
0x03	U8	1	Bits by pixel =1
0x04	U16	2	*W* width of the bitmap, little endian, (must be a multiple of 16)
0x06	U8	1	*H* height of the bitmap
0x09	U8	1	Base line (starting from the bottom)
0x07	U8	1	First defined character
0x08	U8	1	Last defined character
0x0A	U16[]	2*N	Coordinates of the last column of each character (little endian)
0x0A +2*N	U8[]	*H** *W* / 16	Bitmap data

Practical example

Let us imagine that we wish to define a lower case 3x5 pixel font for numbers 0 to 9. The bitmap would be the following:



Font bitmap example

The font file contains the following data:

```
Offset      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000000   59 46 01 01 30 00 06 01  30 39 03 00 05 00 09 00
00000010   0D 00 11 00 15 00 19 00  1D 00 21 00 25 00 EB BA
00000020   3B BB B8 00 A8 8A 22 0A  A8 00 AB BA BB 8B B8 00
00000030   AA 0B 8A 8A 88 00 EB B8  BB 8B B8 00 00 00 00 00
00000040   00 00
```

Note that the bitmap width must be a multiple of 16 pixels and that the height cannot be above 255 pixels. Moreover, blank spaces between two characters are encoded directly in the image. You do not have to leave a blank space below the characters if you do not intend to use your font in console mode.

You can find in the Delphi library a small Windows tool[2] enabling you to generate font files from the system fonts.

3.7. Sequences and animations

You can pre-program animations and play them as background tasks. To do so, you must call the *newSequence* method of the *Display* object, then call the available graphical methods. You can insert waiting times with *pauseSequence*. When you are done recording the sequence components, call *saveSequence*. The sequence is then saved in the Yocto-MaxiDisplay-G file system. You can play it back at will with *playSequence*. You can create loops by calling *playSequence* within a sequence.

You can find in the libraries a code sample[3] illustrating how sequences work. As soon as this example is run, the screen starts playing the sequence indefinitely.

Be aware that sequences change the values of the layer parameters (current point, current color, etc.) with which they work. If you use a sequence as an animation in background task, make sure to work with a different layer than the one used by your sequence.

Startup sequence

When your Yocto-MaxiDisplay-G is powered on, it runs the *yocto.seq* sequence which is hard-coded inside the module. But you can configure your module so that it runs a sequence that you have chosen.

3.8. Optimizations

While the Yocto-MaxiDisplay-G has its own processor and offers numerous graphical routines, it stays a relatively slow system compared to a classic display system. This slowness is the result of using the HID protocol so that the Yocto-MaxiDisplay-G can be driven without drivers. The transfer

[2] Examples\Display-font-generator
[3] Prog-Display-Sequences

rate between the screen and the computer is limited to 64Ko per second. Each request takes up about 3ms. However, there are techniques to optimize display.

Writing in hidden layers

Actions to be performed on visible layers are immediately sent to the screen in order to be executed as soon as possible. On the other hand, actions to be performed on hidden layers are buffered, in order to send several commands at once. It is therefore much more efficient to write in a hidden layer and to make it visible afterwards, which makes double buffering a very interesting technique.

Double buffering

The technique called *double buffering* enables you to display animations without making visible artifacts linked to the creation of the graphics. It consists in working on two layers, one visible, the other one hidden. The images are created in the hidden layer, and once the image is complete, the two layers are switched. In the libraries, you can find an example[4] using this technique to animate a Von Koch flake.

Using a bitmap

When the graphics become complex, it becomes more efficient to compute a bitmap on the host computer and to send it to the screen. Use the *drawBitmap* to do this. Bitmap data are encoded in a byte array, line by line, starting from the top left corner. In each byte, the most significant bit represents the leftmost pixel. In the libraries, you can find an example[5] computing a Mandelbrot set based on this principle.

Here is the C code allowing you to draw a pixel at the *(x,y)* coordinates in the byte array representing a *w* x *h* bitmap.

```
void putpixel(unsigned char *data, int x, int y)
{
    int bytesPerLine = (w + 7) >> 3;
    data[ (x >> 3) + (y * bytesPerLine) ] |= 128 >> (x & 7);
}
```

[4] Prog-Display-DoubleBuffering
[5] Prog-Display-DrawBitmap

4. The embedded file system

Your Yocto-MaxiDisplay-G contains a small embedded file system, allowing it to store personalized files for its own use. You can manipulate the file system thanks to the *yocto_files* library.

4.1. Usage

Interactive usage with the VirtualHub

The *VirtualHub* provides a succinct interface to manipulate the content of the file system: simply click the *configuration* button corresponding to your module in the *VirtualHub* interface, then the *manage files* button. The files are listed and you can view them, erase them, or add new ones (downloads).

Because of its small size, the file system does not have an explicit concept of directories. You can nevertheless use the slash sign "/" inside file names to sort them as if they were in directories.

Programmed usage

Use the *yocto_files* library to manage the file system. Basic functions are available:

- *upload* creates a new file on the module, with a content that you provide;
- *get_list* lists the files on the module, including their content size and CRC32;
- *download* retrieves in a variable the content of a file present on the module;
- *remove* erases a file from the module;
- *format* resets the file system to an empty, not fragmented state.

A piece of software using a well designed file system should always start by making sure that all the files necessary for its working are available on the module, and if needed upload them on the module. We can thus transparently manage software updates and application deployment on new modules. To make file versions easier to detect, the *get_list* method returns for each file a 32 bit signature called CRC (Cyclic Redundancy Check) which identifies in a reliable manner the file content. Thus, if the file CRC corresponds, there is less than one chance over 4 billions that the content is not the correct one. You can even compute in advance in your software the CRC of the content you want, and therefore check it without having to download the files. The CRC function used by the Yoctopuce file system is the same as Ethernet, Gzip, PNG, etc. Its characteristic value for the nine character string "123456789" is 0xCBF43926.

HTTP usage

You can access the files that your have downloaded on your Yocto-MaxiDisplay-G by HTTP, at the root of the module (at the same level as the REST API). This allows you to load personalized HTML

and Javascript interface pages, for example. You cannot, however, replace the content of a file preloaded on the module, you can only add new ones.

4.2. Limitations

The file system embedded on your Yocto-MaxiDisplay-G has some technical limitations:

- Its maximal storage space is 3.5MB, allocated in blocks enabling to store up to about 128 files.
- Erasing a file does not necessarily immediately free all the space used by the file. The non freed space is completely reused if you create a new file with the same name, but not necessarily if you create files with a distinct name each time. For this reason, it is not recommended to automatically create files with ever changing names.
- You can recover the whole non freed space with the *format* command which frees all the files.
- Each firmware update implicitly provokes a complete reformatting of the file system.
- As all flash memories, the memory used to store the files has a life of about 100'000 erasing cycles. It is enough, but it is not infinite. Make sure that you do not write and erase files uselessly and very quickly in a loop, or you may destroy your module.

5. 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.

5.1. Prerequisites

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

A computer

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

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

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

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

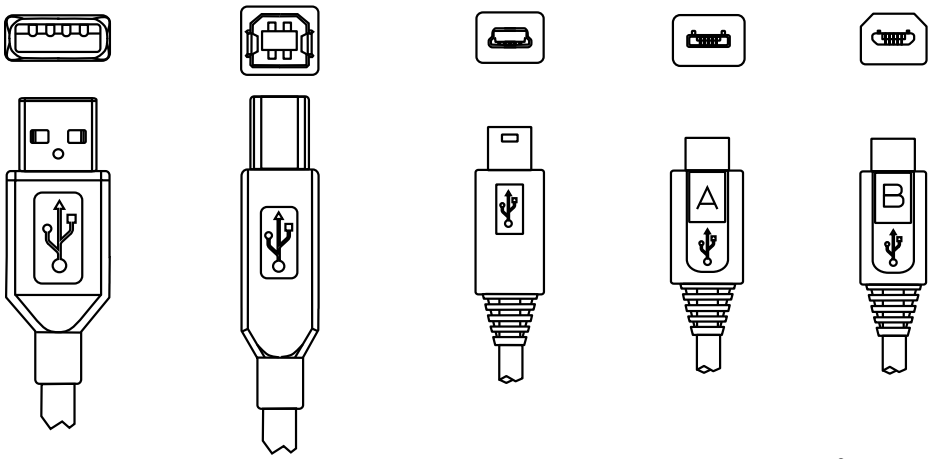
Linux kernels currently supported are the 2.6 branch, the 3.x branch and the 4.x branch. Other versions of the Linux kernel, and even other UNIX variants, are very likely to work as well, as Linux support is implemented through the standard **libusb** API. Yoctopuce is frequently testing its modules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

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

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

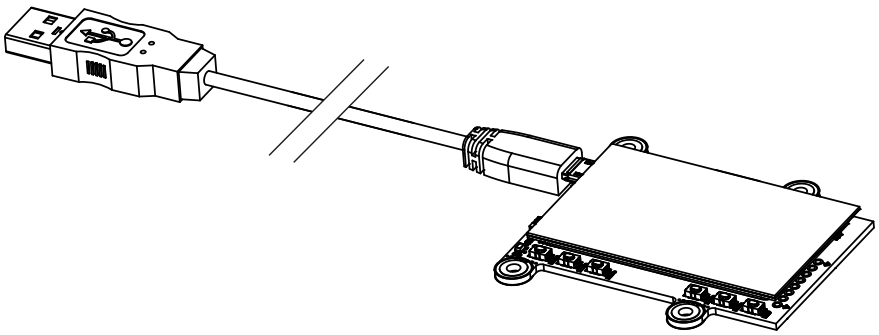
A USB 2.0 cable, type A-micro B

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



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

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



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

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

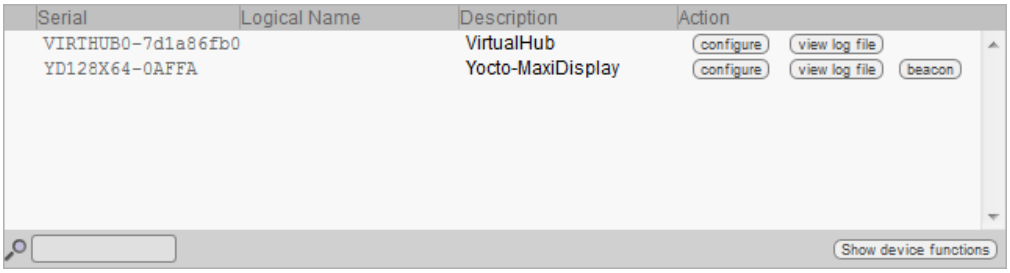
5.2. Testing USB connectivity

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

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

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

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.



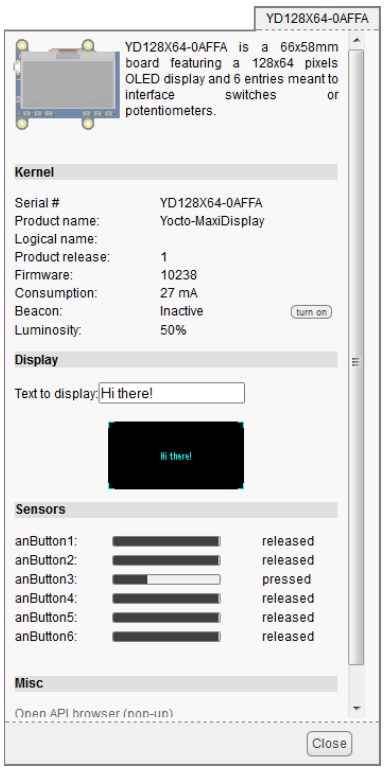
Module list as displayed in your web bowser

5.3. 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.

5.4. 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-MaxiDisplay-G.

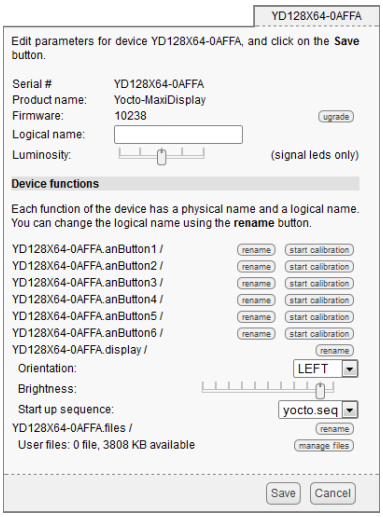


Properties of the Yocto-MaxiDisplay-G module

This window allows you, among other things, to display an arbitrary text on the screen and to check the state of the anButtons inputs.

5.5. Configuration

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



Yocto-MaxiDisplay-G module configuration.

Firmware

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

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

Logical name of the module

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

Luminosity

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

Logical names of functions

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

The functions provided by the Yocto-MaxiDisplay-G module are *display*, corresponding to the screen, *anButton1* to *anButton6* to manage the potentiometer inputs, and *files* corresponding to the file system.

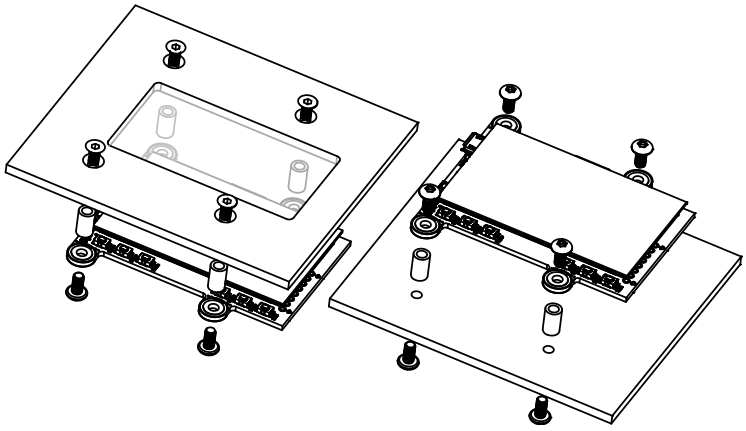
[5] www.yoctopuce.com/EN/virtualhub.php
[6] More information available in the virtual hub documentation

6. Assembly and connections

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

6.1. 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-MaxiDisplay-G module contains 3mm assembly holes. You can use these holes for screws. The screw head diameter must not be larger than 8mm or they will damage the module circuits. Make sure that the lower surface of the module is not in contact with the support. We recommend using spacers, but other methods are possible. Nothing prevents you from fixing the module with a glue gun; it will not be good-looking, but it will hold.

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

6.2. USB power distribution

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

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

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

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

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

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

6.3. Electromagnetic compatibility (EMI)

Connection methods to integrate the Yocto-MaxiDisplay-G 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.

[1] 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.

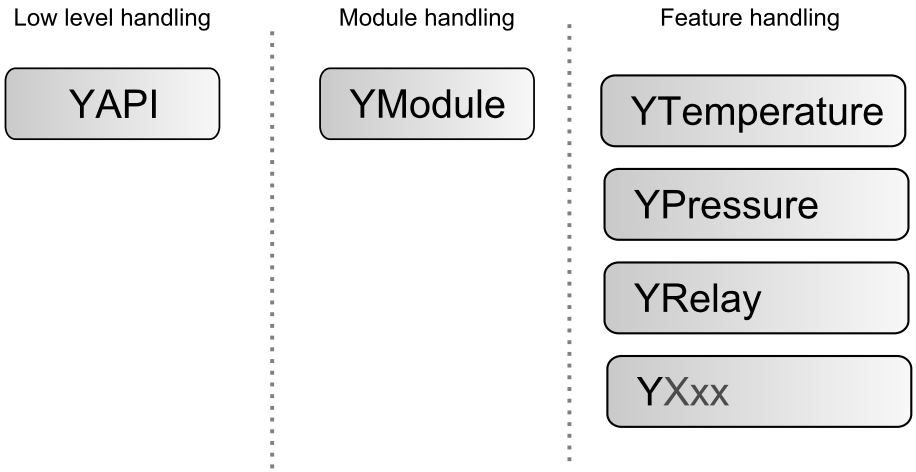
7. Programming, general concepts

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

7.1. Programming paradigm

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

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



Structure of the Yoctopuce API.

The YSensor class

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

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

Programmation

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

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

Access to the functions of a module

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

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

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

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

Function handling

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

Access to the modules

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

Functions/Module interaction

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

7.2. The Yocto-MaxiDisplay-G module

The Yocto-MaxiDisplay-G is a 128x64 OLED display. It includes a flash-based filesystem to store files (such as images, fonts and animated sequences) and six instances of the AnButton function, corresponding to the six analog inputs (potentiometer or button reading) 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

display : Display

attribute	type	modifiable ?
logicalName	String	modifiable
advertisedValue	String	modifiable
enabled	Boolean	modifiable
startupSeq	String	modifiable
brightness	0..100%	modifiable
orientation	Enumerated	modifiable
displayWidth	Integer	read-only
displayHeight	Integer	read-only
displayType	Enumerated	read-only
layerWidth	Integer	read-only
layerHeight	Integer	read-only
layerCount	Integer	read-only
command	String	modifiable

files : Files

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

anButton1 : AnButton
anButton2 : AnButton
anButton3 : AnButton
anButton4 : AnButton
anButton5 : AnButton
anButton6 : AnButton

attribute	type	modifiable ?
logicalName	String	modifiable
advertisedValue	String	modifiable
calibratedValue	Integer	read-only
rawValue	Integer	read-only
analogCalibration	On/Off	modifiable
calibrationMax	Integer	modifiable
calibrationMin	Integer	modifiable
sensitivity	Integer	modifiable
isPressed	Boolean	read-only
lastTimePressed	Time	read-only
lastTimeReleased	Time	read-only
pulseCounter	Integer	modifiable
pulseTimer	Time	read-only
inputType	Enumerated	modifiable

7.3. Module

Global parameters control interface for all Yoctopuce devices

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

productName

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

serialNumber

Character string containing the serial number, unique and programmed at the factory. For a Yocto-MaxiDisplay-G module, this serial number always starts with YD128G64. 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 74 at the factory.

productRelease

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

firmwareRelease

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

persistentSettings

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

luminosity

Lighting strength of the informative leds (e.g. the Yocto-Led) contained in the module. It is an integer value which varies between 0 (LEDs turned off) and 100 (maximum led intensity). The default value

is 50. To change the strength of the module LEDs, or to turn them off completely, you only need to change this value.

beacon

Activity of the localization beacon of the module.

upTime

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

usbCurrent

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

rebootCountdown

Countdown to use for triggering a reboot of the module.

userVar

32bit integer variable available for user storage.

7.4. Display

display control interface, available for instance in the Yocto-Display, the Yocto-MaxiDisplay, the Yocto-MaxiDisplay-G or the Yocto-MiniDisplay

The `YDisplay` class allows to drive Yoctopuce displays. Yoctopuce display interface has been designed to easily show information and images. The device provides built-in multi-layer rendering. Layers can be drawn offline, individually, and freely moved on the display. It can also replay recorded sequences (animations). In order to draw on the screen, you should use the `display.get_displayLayer` method to retrieve the layer(s) on which you want to draw, and then use methods defined in `YDisplayLayer` to draw on the layers.

logicalName

Character string containing the logical name of the display, 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 display directly. If two displays 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 display, that is automatically advertised up to the parent hub. For a display, the advertised value is its power state (ON or OFF).

enabled

Power state of the display. The display can be enabled and disabled at will using this attribute.

startupSeq

Name of the sequence to play when the display is powered on.

brightness

Brightness of the display. It is an integer value which varies between 0 (very dark display) and 100 (very bright display).

orientation

Display orientation. The orientation is defined as the side of the screen where the USB connector is located when the display is up straight.

displayWidth

Display width, in pixels.

displayHeight

Display height, in pixels.

displayType

Display type: monochrome (MONO), gray levels (GRAY) or full color (RGB).

layerWidth

Width of the layers to draw on, in pixels.

layerHeight

Height of the layers to draw on, in pixels.

layerCount

Available layers to draw on.

command

Magic attribute used to send content to the display. If a command is not interpreted as expected, check the device logs.

7.5. AnButton

analog input control interface, available for instance in the Yocto-Buzzer, the Yocto-Knob, the Yocto-MaxiBuzzer or the Yocto-MaxiDisplay

The `YAnButton` class provide access to basic resistive inputs. Such inputs can be used to measure the state of a simple button as well as to read an analog potentiometer (variable resistance). This can be use for instance with a continuous rotating knob, a throttle grip or a joystick. The module is capable to calibrate itself on min and max values, in order to compute a calibrated value that varies proportionally with the potentiometer position, regardless of its total resistance.

logicalName

Character string containing the logical name of the analog input, 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 analog input directly. If two analog inputs 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 analog input, that is be automatically advertised up to the parent hub. For an analog input, the advertised value is the calibrated measured value (a number between 0 and 1000).

calibratedValue

Calibrated value of the anlog input, as an integer from 0 to 1000, included. If no calibration has been run, the calibrated value is simply the measured value, scaled down to the range 0...1000, without linearity correction.

rawValue

Measured value of the analog input, as is, formatted as an integer from 0 to 4095. The measured value is zero when the input resistance is zero (closed contact), and reaches 4095 as the input

resistance moves toward infinity (open contact). Be aware that this value does not vary linearly with regard to the input resistance (and to the known position). If you need a linear value, perform a calibration and use the computed value `calibratedValue`.

analogCalibration

Use this attribute to start and stop the calibration process for the analog input. When the calibration is enabled, the module records the minimal and maximal measured values into `calibrationMin` and `calibrationMax`. Once the calibration is over (stopped), the module can automatically compute a running calibrated value for the every measure, varying linearly with the measured resistance value.

calibrationMax

Maximal observed raw measure during calibration. You can also change this value by software if you want to force a theoretical calibration.

calibrationMin

Minimal observed raw measure during calibration. You can also change this value by software if you want to force a theoretical calibration.

sensitivity

Sensitivity of the analog input to trigger user callbacks. The sensitivity corresponds to the difference of value required to propagate a new advertised value and trigger the corresponding user callbacks. If the value is too small, it could cause spurious callbacks if the measured input is not stable enough.

isPressed

Logical state of the input, considered as a binary input (on/off button). The logical state is pressed when the contact is closed, and non-pressed when the contact is open. The module implements a slight averaging and a Schmitt trigger to get a more reliable binary reading.

lastTimePressed

Absolute time of the last "button pressed" event on the input (the input contact transitioned from open to closed). The time reference is the same as the `upTime` attribute, i.e. the elapsed time since the module was turned on.

lastTimeReleased

Absolute time of the last "button released" event on the input (the input contact transitioned from closed to open). The time reference is the same as the `upTime` attribute, i.e. the elapsed time since the module was turned on. If you subtract from this value the `lastTimePressed` value, you can get the duration of the last button pressure.

pulseCounter

Pulse counter (32 bits), incremented each time the button state changes (PRESSED / RELEASED), Each pulse therefore increment the counter by 2. This counter is set to zero when resetCounter() is called and each time the device restarts

pulseTimer

Elapsed time since last pulse counter reset (milliseconds).

inputType

Type of knob connected to the input (analog or multiplexed binary switches)

7.6. Files

filesystem control interface, available for instance in the Yocto-Color-V2, the Yocto-Serial, the YoctoHub-Ethernet or the YoctoHub-Wireless-n

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

logicalName

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

advertisedValue

Short character string summarizing the current state of the filesystem, that is automatically advertised up to the parent hub. For a filesystem, the advertised value is the number of files loaded in the filesystem.

filesCount

Number of files currently loaded in the filesystem.

freeSpace

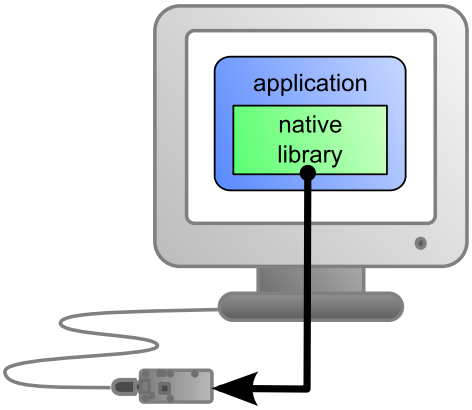
Free space for uploading new files to the filesystem, in bytes.

7.7. What interface: Native, DLL or Service ?

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

Native control

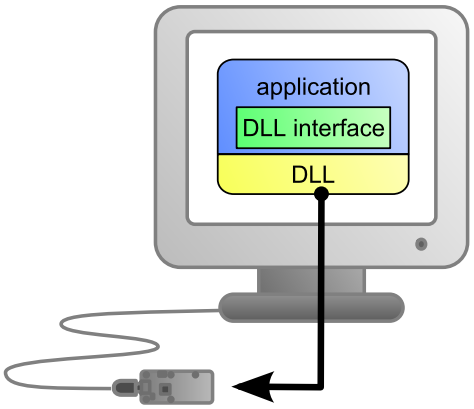
In this case, the software driving your project is compiled directly with a library which provides control of the modules. Objectively, it is the simplest and most elegant solution for the end user. The end user then only needs to plug the USB cable and run your software for everything to work. Unfortunately,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



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

Native control by DLL

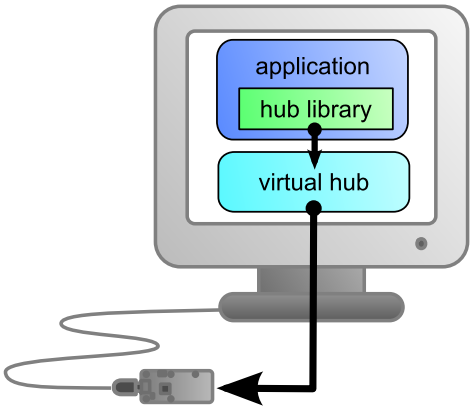
Here, the main part of the code controlling the modules is located in a DLL. The software is compiled with a small library which provides control of the DLL. It is the fastest method to code module support in a given language. Indeed, the "useful" part of the control code is located in the DLL which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



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

Control by service

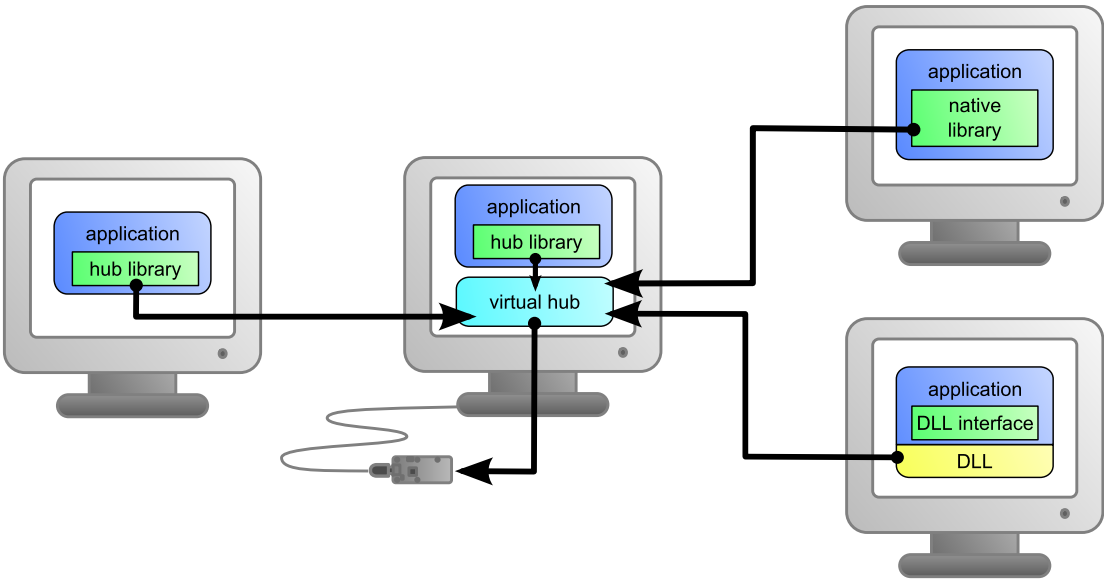
Some languages do simply not allow you to easily gain access to the hardware layers of the machine. It is the case for Javascript, for instance. To deal with this case, Yoctopuce provides a solution in the form of a small piece of software called *VirtualHub*[1]. It can access the modules, and your application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



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

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

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



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

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

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

Support methods for different languages

Limitations of the Yoctopuce libraries

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

7.8. Programming, where to start?

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

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

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

8. Using the Yocto-MaxiDisplay-G in command line

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

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

8.1. Installing

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

UNABLE TO INCLUDE
http://172.17.17.77/tu/projects/yoctodisplay-128x64-G/public/examples/cmdline/helloworld.cmd

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

8.2. Use: general description

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

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

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

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

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

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

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

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

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

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

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

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

8.3. Control of the Display function

To control the Display function of your Yocto-MaxiDisplay-G, you need the YDisplay executable file.

For instance, you can launch:

UNABLE TO INCLUDE http://172.17.17.77/tu/projects/yoctodisplay-128x64-G/public/examples/cmdline/helloworld.cmd

This example uses the "*any*" target to indicate that we want to work on the first Display 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-MaxiDisplay-G module with the *YD128G64-123456* serial number which you have called "*MyModule*", and its display 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:\>YDisplay YD128G64-123456.display describe
C:\>YDisplay YD128G64-123456.MyFunction describe
C:\>YDisplay MyModule.display describe
C:\>YDisplay MyModule.MyFunction describe
C:\>YDisplay MyFunction describe
```

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

```
C:\>YDisplay all describe
```

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

```
C:\>YDisplay /help
```

8.4. Control of the module part

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

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

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

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

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

```
C:\>YModule YD128G64-12346 set_logicalName MonPremierModule

C:\>YModule YD128G64-12346 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 YD128G64-12346 set_logicalName MonPremierModule
C:\>YModule YD128G64-12346 saveToFlash
```

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

```
C:\>YModule -s  YD128G64-12346 set_logicalName MonPremierModule
```

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

8.5. Limitations

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

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

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

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

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

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

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

9. Using the Yocto-MaxiDisplay-G with Python

Python is an interpreted object oriented language developed by Guido van Rossum. Among its advantages is the fact that it is free, and the fact that it is available for most platforms, Windows as well as UNIX. It is an ideal language to write small scripts on a napkin. The Yoctopuce library is compatible with Python 2.6+ and 3+. It works under Windows, Mac OS X, and Linux, Intel as well as ARM. The library was tested with Python 2.6 and Python 3.2. Python interpreters are available on the Python web site[1].

9.1. 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.

9.2. Dynamic library

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

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

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

9.3. Control of the Display function

A few lines of code are enough to use a Yocto-MaxiDisplay-G. Here is the skeleton of a Python code snipplet to use the Display 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
display = YDisplay.FindDisplay("YD128G64-123456.display")

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

[...]
```

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.

YDisplay.FindDisplay

The `YDisplay.FindDisplay` function allows you to find a display 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-MaxiDisplay-G module with serial number *YD128G64-123456* which you have named "*MyModule*", and for which you have given the *display* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
display = YDisplay.FindDisplay("YD128G64-123456.display")
display = YDisplay.FindDisplay("YD128G64-123456.MyFunction")
display = YDisplay.FindDisplay("MyModule.display")
display = YDisplay.FindDisplay("MyModule.MyFunction")
display = YDisplay.FindDisplay("MyFunction")
```

`YDisplay.FindDisplay` returns an object which you can then use at will to control the display.

isOnline

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

get_displayLayer

The `get_displayLayer()` method of the object returned by `YDisplay.FindDisplay` allows you to retrieve the object corresponding to one of the screen layers. This object implements all the graphical routines.

A real example

Launch Python and open the corresponding sample script provided in the directory **Examples/Doc-GettingStarted-Yocto-MaxiDisplay-G** 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.

UNABLE TO INCLUDE
http://172.17.17.77/tu/projects/yoctodisplay-128x64-G/public/examples/python/helloworld.py

9.4. Control of the module part

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

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

from yocto_api import *


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


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

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

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

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

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

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

Changing the module settings

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

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

from yocto_api import *


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


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

```
    usage()

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

m = YModule.FindModule(sys.argv[1])  # use serial or logical name
if m.isOnline():
    newname = sys.argv[2]
    if not YAPI.CheckLogicalName(newname):
        sys.exit("Invalid name (" + newname + ")")
    m.set_logicalName(newname)
    m.saveToFlash()  # do not forget this
    print("Module: serial= " + m.get_serialNumber() + " / name= " + m.get_logicalName())
else:
    sys.exit("not connected (check identification and USB cable")
YAPI.FreeAPI()
```

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

Listing the modules

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

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


from yocto_api import *

errmsg = YRefParam()

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

print('Device list')

module = YModule.FirstModule()
while module is not None:
    print(module.get_serialNumber() + ' (' + module.get_productName() + ')')
    module = module.nextModule()
YAPI.FreeAPI()
```

9.5. Error handling

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

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

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

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

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

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

10. Using Yocto-MaxiDisplay-G with C++

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

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

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

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

10.1. Control of the Display function

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

```
#include "yocto_api.h"
#include "yocto_display.h"

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

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

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

```
YDisplay *display;
display = YDisplay::FindDisplay("YD128G64-123456.display");

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

Let's look at these lines in more details.

yocto_api.h et yocto_display.h

These two include files provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api.h` must always be used, `yocto_display.h` is necessary to manage modules containing a display, such as Yocto-MaxiDisplay-G.

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.

YDisplay::FindDisplay

The `YDisplay::FindDisplay` function allows you to find a display 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-MaxiDisplay-G module with serial number *YD128G64-123456* which you have named "*MyModule*", and for which you have given the *display* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
YDisplay *display = YDisplay::FindDisplay("YD128G64-123456.display");
YDisplay *display = YDisplay::FindDisplay("YD128G64-123456.MyFunction");
YDisplay *display = YDisplay::FindDisplay("MyModule.display");
YDisplay *display = YDisplay::FindDisplay("MyModule.MyFunction");
YDisplay *display = YDisplay::FindDisplay("MyFunction");
```

`YDisplay::FindDisplay` returns an object which you can then use at will to control the display.

isOnline

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

get_displayLayer

The `get_diplayLayer()` method of the object returned by `YFindDisplay` allows you to retrieve the object corresponding to one of the screen layers. This object implements all the graphical routines.

A real example

Launch your C++ environment and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-MaxiDisplay-G** 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.

UNABLE TO INCLUDE
http://172.17.17.77/tu/projects/yoctodisplay-128x64-G/public/examples/C++/helloworld.cpp

10.2. Control of the module part

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

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

#include "yocto_api.h"

using namespace std;

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


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

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

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

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

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

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

Changing the module settings

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

forget its current settings by using the `revertFromFlash()` method. The short example below allows you to modify the logical name of a module.

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

#include "yocto_api.h"

using namespace std;

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

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

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

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

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

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

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

Listing the modules

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

```
#include <iostream>

#include "yocto_api.h"

using namespace std;

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

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

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

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

10.3. Error handling

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

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

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

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

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

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

10.4. Integration variants for the C++ Yoctopuce library

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

Integration in source format (recommended)

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

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

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

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

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

Integration as a static library

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

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

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

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

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

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

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

Integration as a dynamic library

Integration of the Yoctopuce library as a dynamic library allows you to produce an executable smaller than with the two previous methods, and to possibly update this library, if a patch reveals itself necessary, without needing to recompile the source code of the application. On the other hand, it is an integration mode which systematically requires you to copy the dynamic library on the target

machine where the application will run (**yocto.dll** for Windows, **libyocto.so.1.0.1** for Mac OS X and Linux).

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

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

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

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

With GCC, the command line to compile is simply:

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

11. Using Yocto-MaxiDisplay-G with C#

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

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

11.1. Installation

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

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

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

Configuring a Visual C# project

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

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

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

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

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

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

11.3. Control of the Display function

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

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

// Retrieve the object used to interact with the device
YDisplay display = YDisplay.FindDisplay("YD128G64-123456.display");

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

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.

YDisplay.FindDisplay

The `YDisplay.FindDisplay` function allows you to find a display 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-MaxiDisplay-G module with serial number *YD128G64-123456* which you have named "*MyModule*", and for which you have given the *display* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
display = YDisplay.FindDisplay("YD128G64-123456.display");
display = YDisplay.FindDisplay("YD128G64-123456.MyFunction");
display = YDisplay.FindDisplay("MyModule.display");
display = YDisplay.FindDisplay("MyModule.MyFunction");
display = YDisplay.FindDisplay("MyFunction");
```

`YDisplay.FindDisplay` returns an object which you can then use at will to control the display.

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

isOnline

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

get_displayLayer

The `get_displayLayer()` method of the object returned by `YDisplay.FindDisplay` allows you to retrieve the object corresponding to one of the screen layers. This object implements all the graphical routines.

A real example

Launch Microsoft Visual C# and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-MaxiDisplay-G** 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.

UNABLE TO INCLUDE
http://172.17.17.77/tu/projects/yoctodisplay-128x64-G/public/examples/C-sharp/helloworld.cs

11.4. Control of the module part

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

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


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

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

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


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

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

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

```
        }

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

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

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

Changing the module settings

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

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

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

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

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

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

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

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

```
        }

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

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

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

Listing the modules

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

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

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

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

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

11.5. Error handling

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

The simplest way to work around the problem is the one used in the short examples provided in this chapter: before accessing a module, check that it is online with the `isOnline` function, and then

hope that it will stay so during the fraction of a second necessary for the following code lines to run. This method is not perfect, but it can be sufficient in some cases. You must however be aware that you cannot completely exclude an error which would occur after the call to `isOnline` and which could crash the software. The only way to prevent this is to implement one of the two error handling techniques described below.

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

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

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

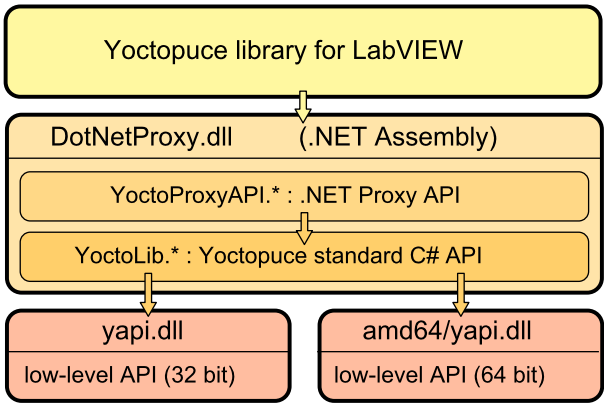
When you work without exceptions, you can obtain an error code and an error message explaining the source of the error. You can request them from the object which returned the error, calling the `errType()` and `errMessage()` methods. Their returned values contain the same information as in the exceptions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

12. Using the Yocto-MaxiDisplay-G with LabVIEW

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

12.1. Architecture

The LabVIEW library is based on the Yoctopuce DotNetProxy library contained in the DotNetProxyLibrary.dll DLL. In fact, it is this DotNetProxy library which takes care 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.

12.2. Compatibility

Firmware

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

LabVIEW for Linux and MacOS

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

LabVIEW NXG

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

About 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.

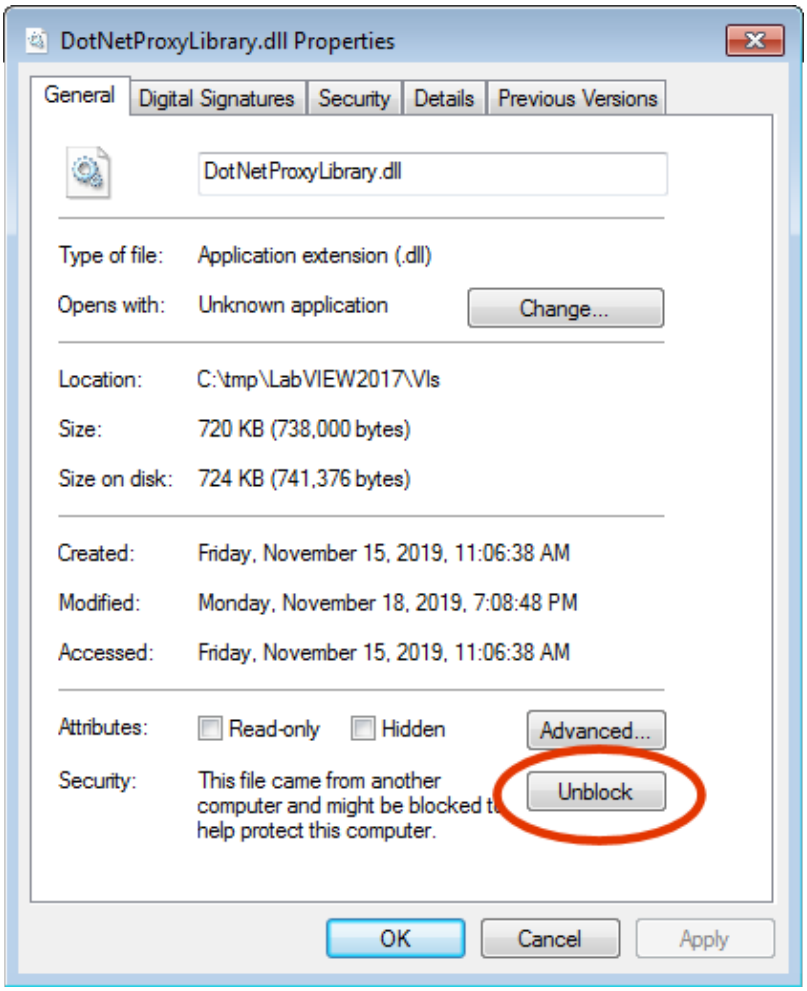
12.3. Installation

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

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

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

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



Unblock the DotNetProxyLibrary DLL.

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

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

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

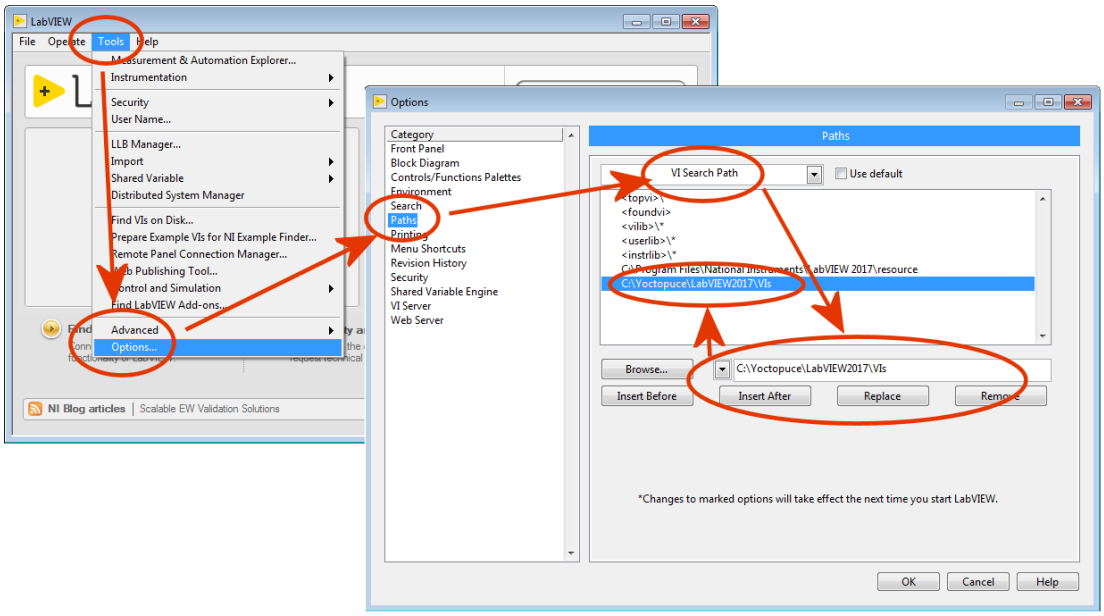
To install the LabVIEW Yoctopuce API, there are several methods.

Method 1 : "Take-out" installation

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

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

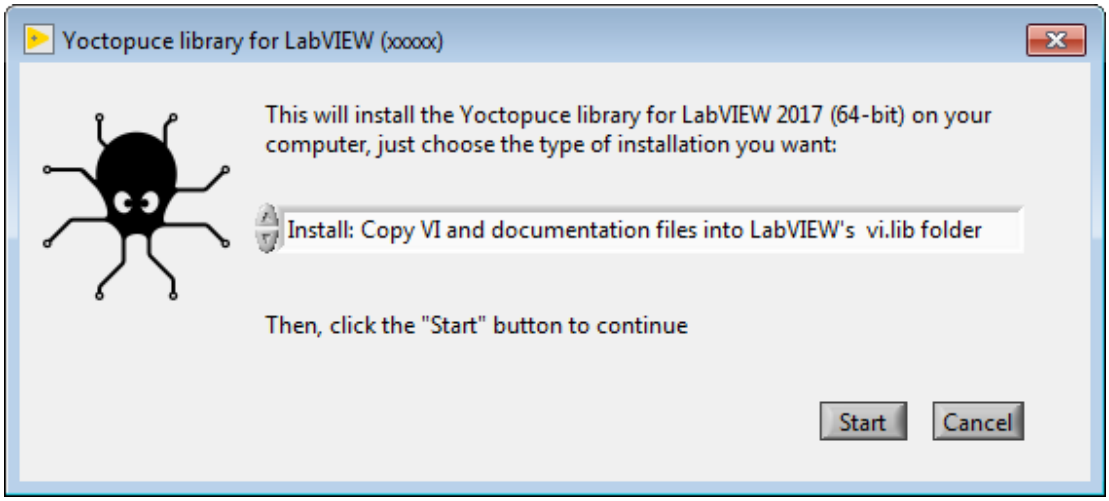
[2] https://knowledge.ni.com/KnowledgeArticleDetails?id=kA00Z000000P8XnSAK



Configuring the "VI Search Path"

Method 2 : Provided installer

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



The provider installer

This installer provide 3 installation options:

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

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

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

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

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

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

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

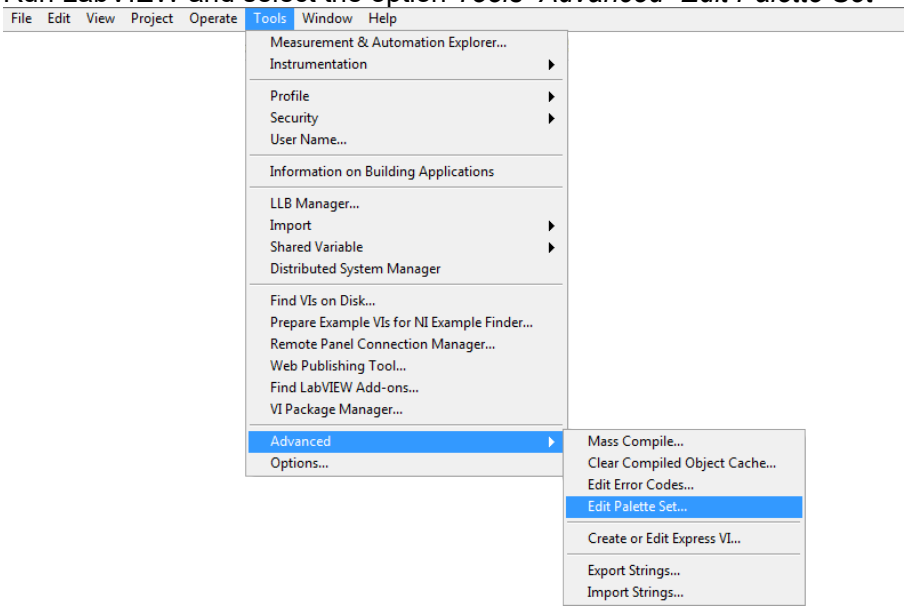
The installer identifies Yoctopuce VIs library folders by checking the presence of the YRegisterHub.vi file in said folders.

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

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

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

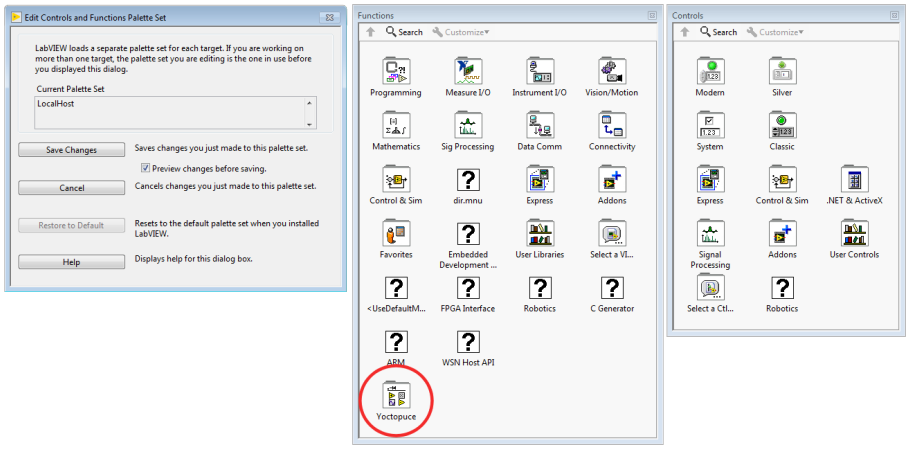
1. Create a *Yoctopuce/API* directory in the *C:\Program Files\National Instruments\LabVIEW xxxx \vi.lib* directory and copy all the VIs and DLLs of the *VIs* directory into it.
2. Create a *Yoctopuce* directory in the *C:\Program Files\National Instruments\LabVIEW xxxx \menus\Categories* directory.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



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

Three windows pop up:

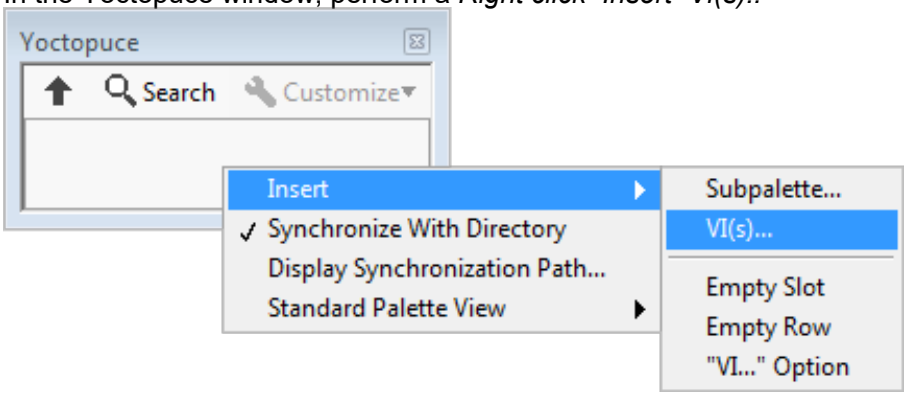
- "Edit Controls and Functions Palette Set"
- "Functions"
- "Controls"



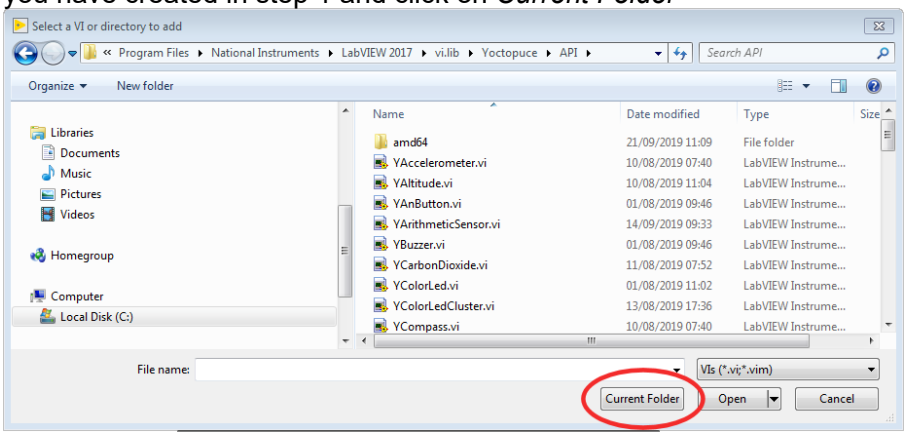
.

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

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

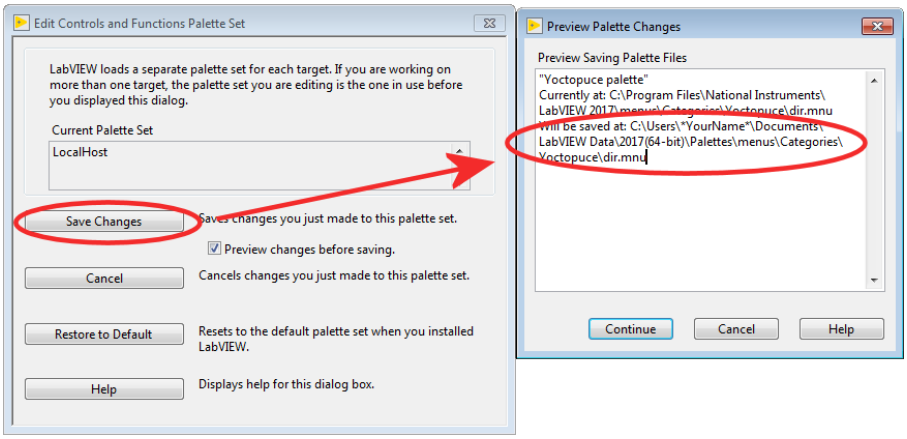


in order to open a file chooser. Put the file chooser in the *vi.lib\Yoctopuce\API* directory that you have created in step 1 and click on *Current Folder*



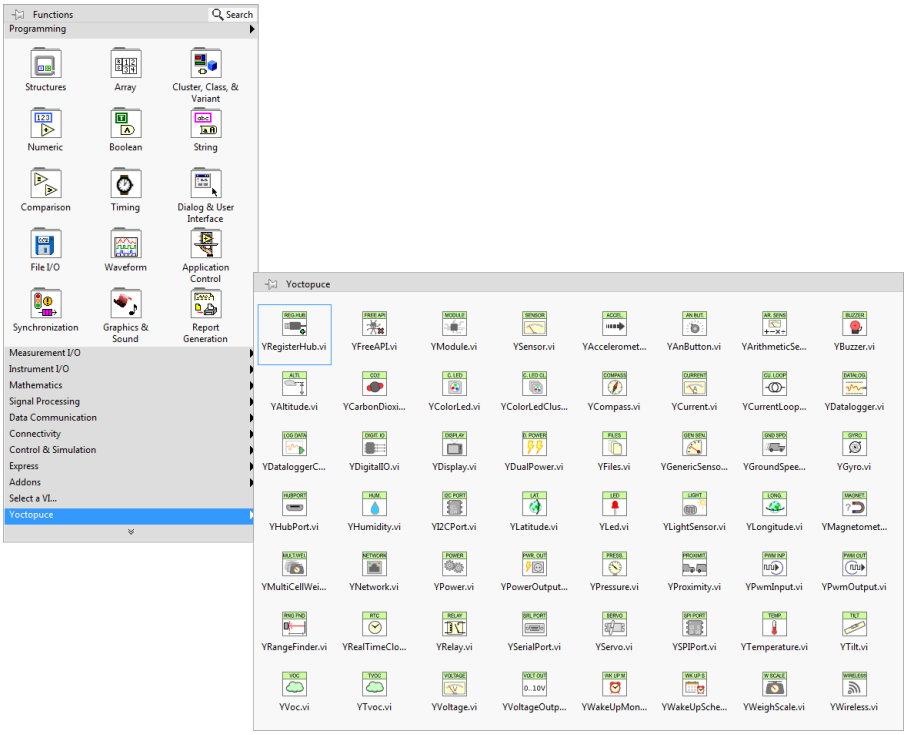
All the Yoctopuce VIs now appear in the Yoctopuce window. By default, they are sorted by alphabetical order, but you can arrange them as you see fit by moving them around with the mouse. For the palette to be easy to use, we recommend to reorganize the icons over 8 columns.

5. In the "*Edit Controls and Functions Palette Set*" window, click on the "Save Changes" button, the window indicates that it has created a `dir.mnu` file in your `Documents` directory.



Copy this file in the "`menus\Categories\Yoctopuce`" directory that you have created in step 2.

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

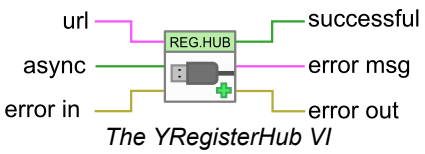


12.4. Presentation of Yoctopuce VIs

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

YRegisterHub

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



The YRegisterHub VI

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

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

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

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

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

YFreeAPI

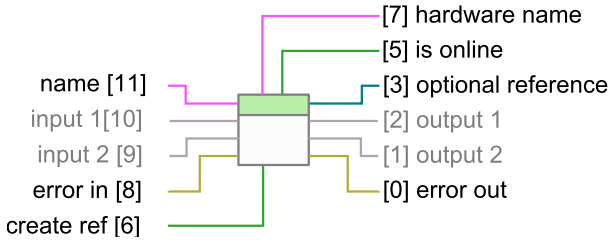
The YFreeAPI VI enables you to free resources allocated by the Yoctopuce API.

The YFreeAPI VI

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

Structure of the VIs corresponding to a class

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



Structure of most VIs of the API.

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

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

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

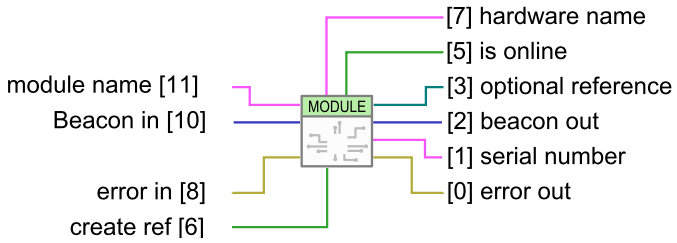
You can find the list of functions available on your Yocto-MaxiDisplay-G in chapter *Programming, general concepts*.

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.

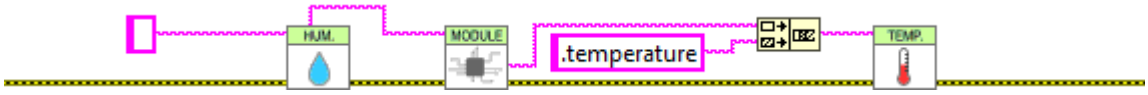
The YModule VI

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

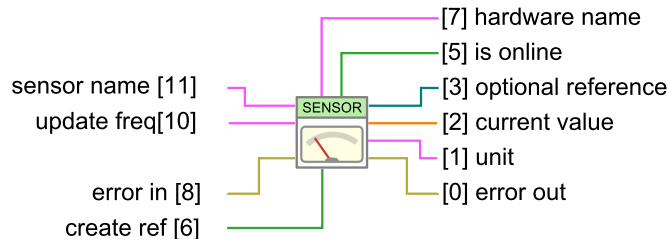
The *name* input works slightly differently from other VIs. If it is called with a *name* parameter corresponding to a function name, the `YModule` VI finds the *Module* function of the module hosting the function. You can therefore easily find the serial number of the module of any function. This enables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



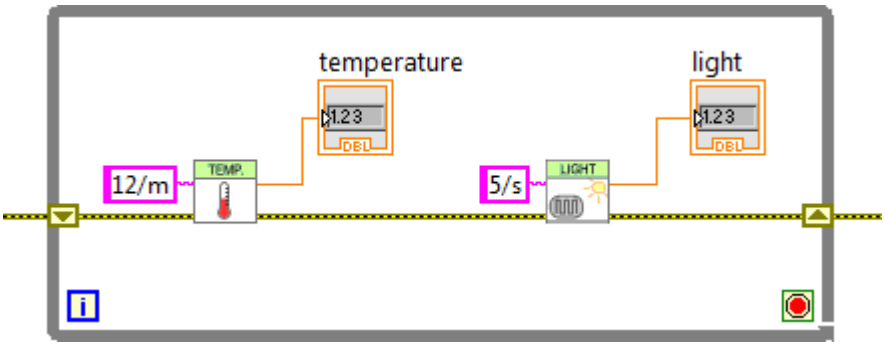
The sensor VIs have all exactly the same geometry

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.

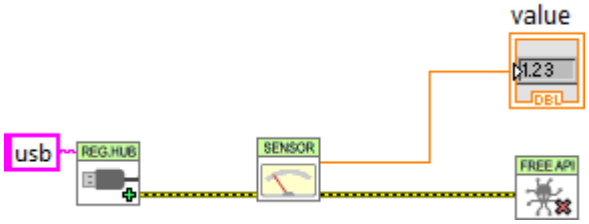


Changing the update frequency of the same module

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.

12.5. Functioning and use of VIs

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

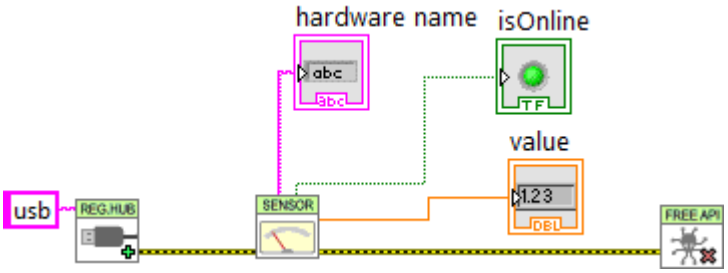


Minimal example of use of the LabVIEW Yoctopuce API

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

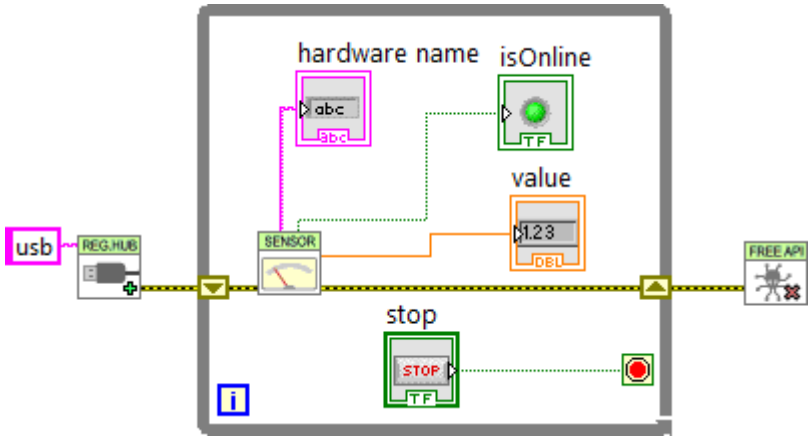
1. It initializes the API in native ("usb") mode with the `YRegisterHub` VI.
2. It displays the value of the first Yoctopuce sensor it finds thanks to the `YSensor` VI.
3. It frees the API thanks to the `YFreeAPI` VI.

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



Use of the hardware name and isOnline outputs

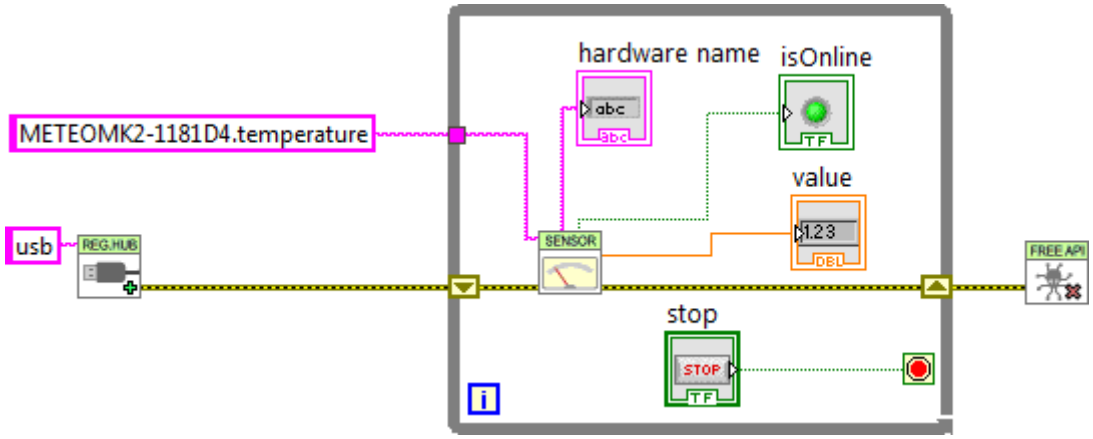
The VIs of the Yoctopuce API are actually an entry door into the library. Internally, this mechanism works independently of the Yoctopuce VIs. Indeed, most communications with electronic modules are managed automatically as background tasks. Therefore, you do not necessarily need to take any specific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



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

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

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

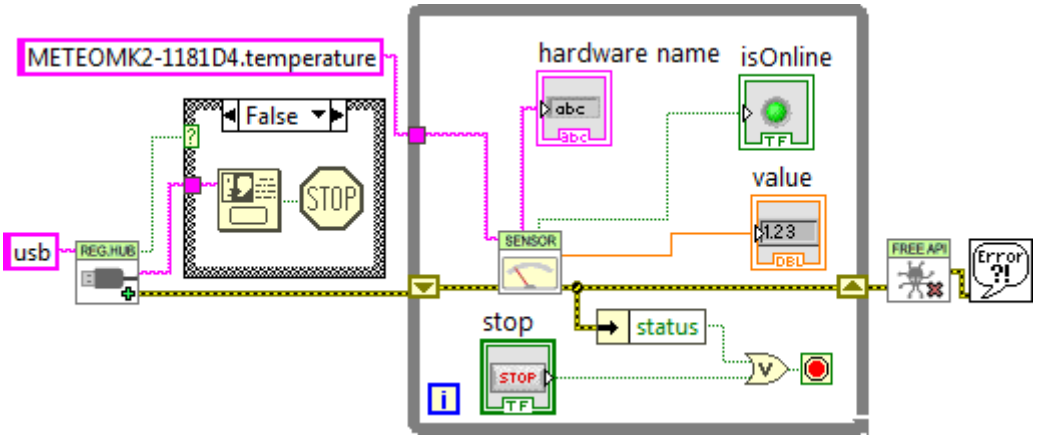


Using names to identify the functions to be used

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.



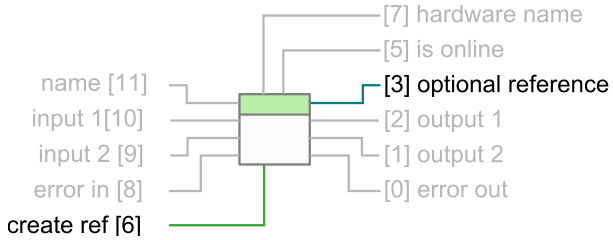
Error handling

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.

12.6. Using *Proxy* objects

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

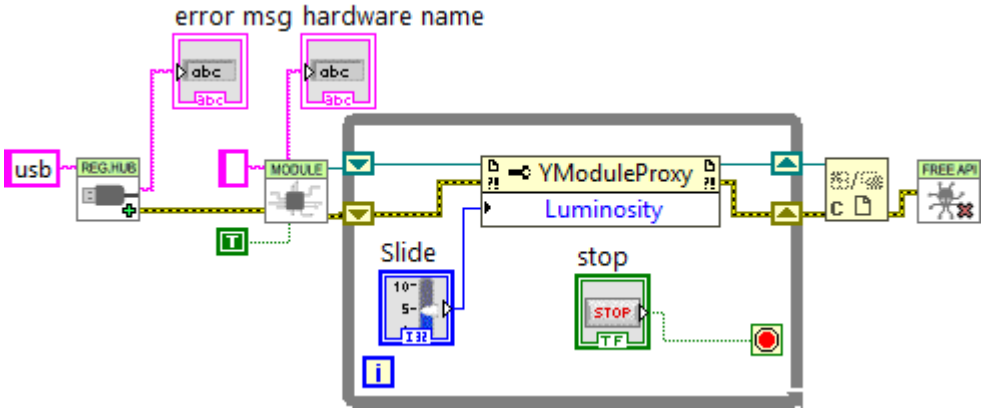
Each VI corresponding to a class has two connectors *create ref* and *optional ref* which enable you to obtain a reference on the *Proxy* object of the *.NET Proxy* API on which the LabVIEW library is built.



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

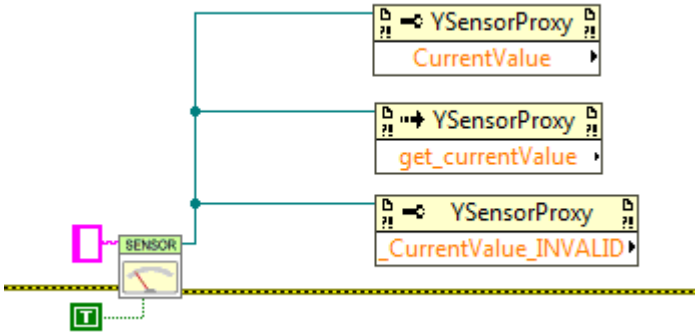
To obtain this reference, you only need to set *optional ref* to TRUE. Note, it is essential to close all references created in this way, otherwise you risk to quickly saturate the computer memory.

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



Regulating the luminosity of the leds of a module

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

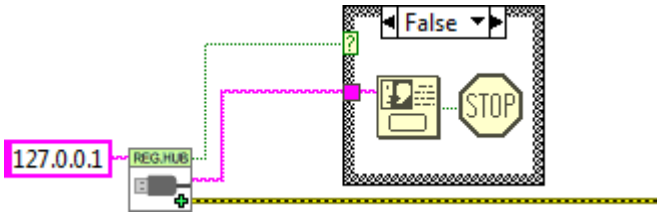


Property and Invoke nodes: Using properties, methods and constants

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

Network mode

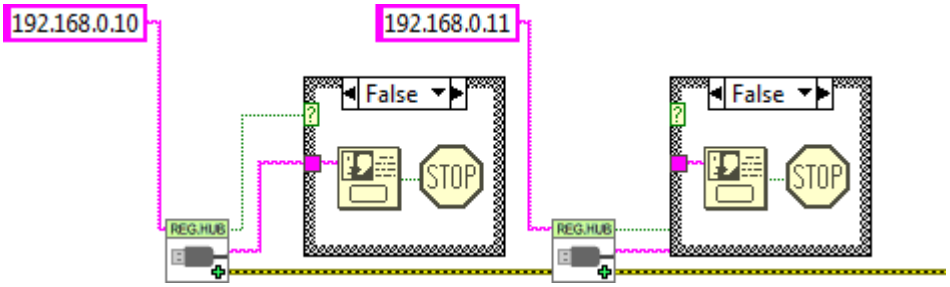
On a given machine, there can be only one process accessing local Yoctopuce modules directly by USB (url set to `"usb"`). It is however possible that multiple process connect in parallel to YoctoHubs[7] or tp a machine on which *VirtualHub*[8] is running, including the local machine. Therefore, if you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



Network mode

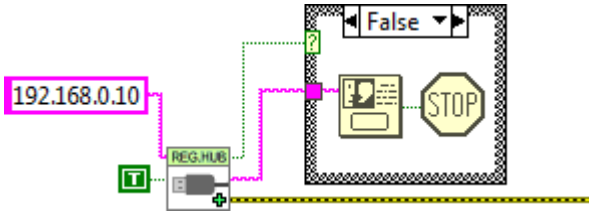
[7] https://www.yoctopuce.com/EN/products/category/extensions-and-networking
[8] www.yoctopuce.com/EN/virtualhub.php

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



You can have multiple network connections

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



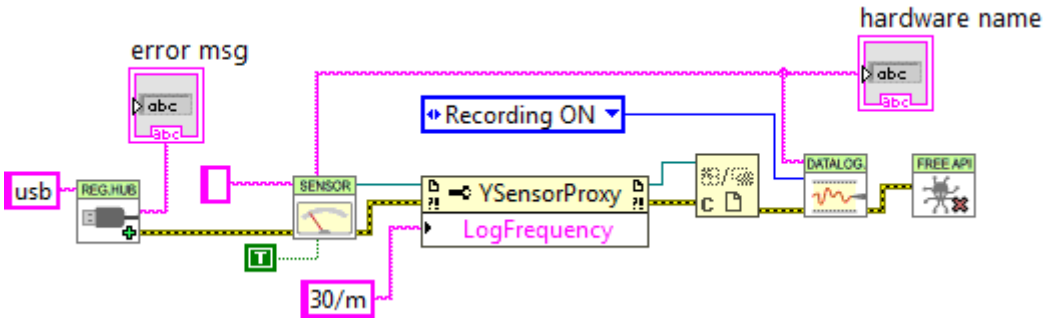
Asynchronous connection

12.7. Managing the data logger

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

Logging

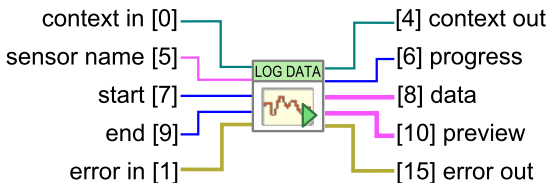
To do so, you must configure the logging frequency by using the "LogFrequency" property which you can reach with a reference on the *Proxy* object of the sensor you are using. Then, you must turn the data logger on thanks to the `YDataLogger` VI. Note that, like with the `YModule` VI, you can obtain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



Activating the data logger

Reading

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

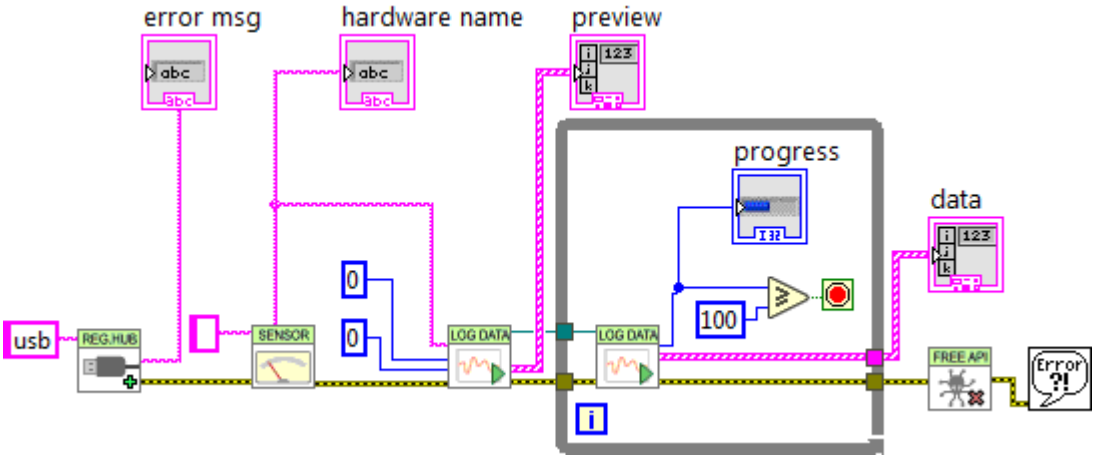


The YDataLoggerContents VI

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

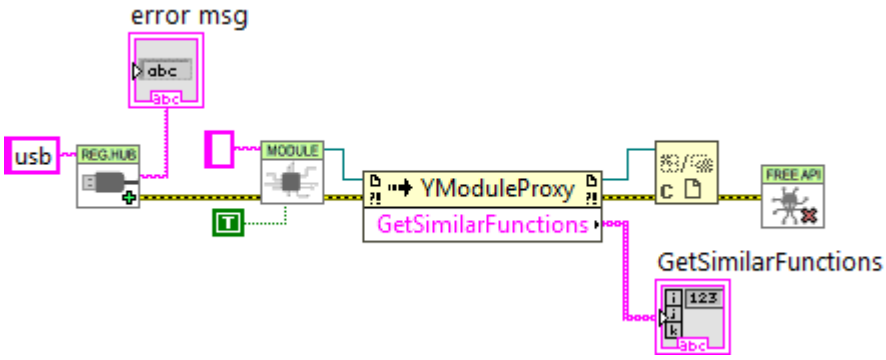
The results and the summary are returned as an array of structures containing the following fields:

- *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.

12.8. Function list

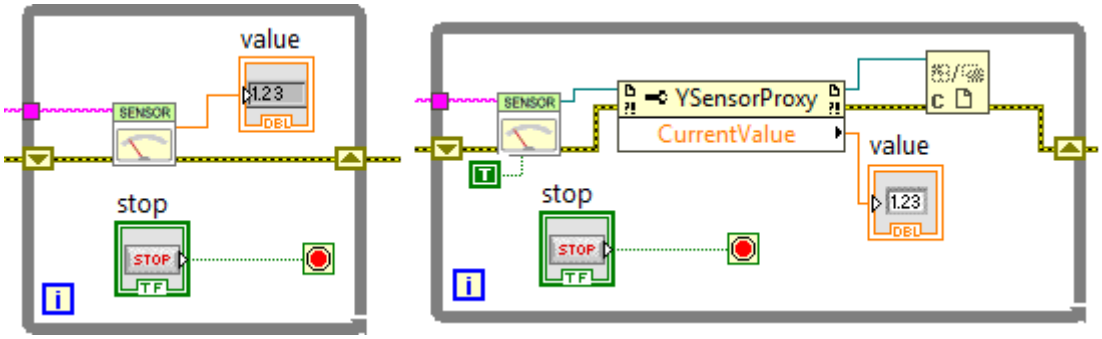
Each VI corresponding to an object of the *Proxy API* enables you to list all the functions of the same class with the *getSimilarfunctions()* method of the corresponding *Proxy* object. Thus, you can easily perform an inventory of all the connected modules, of all the connected sensors, of all the connected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



Retrieving the list of all the modules which are connected

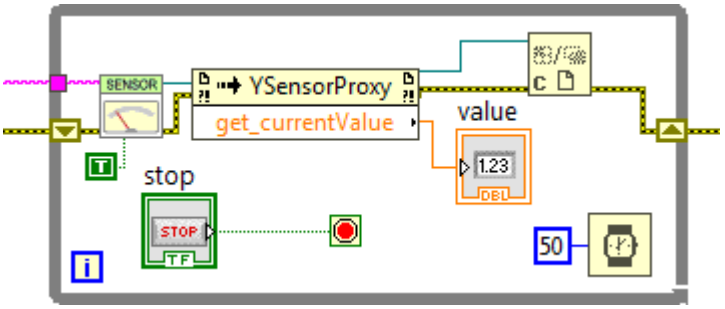
12.9. A word on performances

The LabVIEW Yoctopuce API is optimized so that all the VIs and *.NET Proxy* API object properties generate a minimum of communication with Yoctopuce modules. Thus, you can use them in loops without taking any specific precaution: you *do not have to* slow down the loops with a timer.



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

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



This loop, using a method, must be slowed down

12.10. A full example of a LabVIEW program

UNABLE TO INCLUDE
http://172.17.17.77/tu/projects/yoctodisplay-128x64-G/public/doc-labview-example-EN.html

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

12.11. Differences from other Yoctopuce APIs

Yoctopuce does everything it can to maintain a strong coherence between its different programming libraries. However, LabVIEW being clearly apart as an environment, there are, as a consequence, important differences from the other libraries.

These differences were introduced to make the use of modules as easy as possible and requiring a minimum of LabVIEW code.

YFreeAPI

In the opposite to other languages, you must absolutely free the native API by calling the `YFreeAPI` VI when your code does not need to use the API anymore. If you forget this call, the native API risks to stay locked for the other applications until LabVIEW is completely closed.

Properties

In the opposite to classes of the other APIs, classes available in LabVIEW implement *properties*. By convention, these properties are optimized to generate a minimum of communication with the modules while automatically refreshing. By contrast, methods of type *get_xxx* and *set_xxx* systematically generate communications with the Yoctopuce modules and must be called sparingly.

Callback vs. Properties

There is no callback in the LabVIEW Yoctopuce API, the VIs automatically refresh: they are based on the properties of the *.NET Proxy* API objects.

13. Using the Yocto-MaxiDisplay-G with Java

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

13.1. Getting ready

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

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

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

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

13.2. Control of the Display function

A few lines of code are enough to use a Yocto-MaxiDisplay-G. Here is the skeleton of a Java code snippet to use the Display 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
display = YDisplay.FindDisplay("YD128G64-123456.display");

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

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

```
    // Use display.get_displayLayer()
    [...]
}

[...]
```

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.

YDisplay.FindDisplay

The `YDisplay.FindDisplay` function allows you to find a display 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-MaxiDisplay-G module with serial number *YD128G64-123456* which you have named "*MyModule*", and for which you have given the *display* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
display = YDisplay.FindDisplay("YD128G64-123456.display")
display = YDisplay.FindDisplay("YD128G64-123456.MyFunction")
display = YDisplay.FindDisplay("MyModule.display")
display = YDisplay.FindDisplay("MyModule.MyFunction")
display = YDisplay.FindDisplay("MyFunction")
```

`YDisplay.FindDisplay` returns an object which you can then use at will to control the display.

isOnline

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

get_displayLayer

The `get_displayLayer()` method of the object returned by `YDisplay.FindDisplay` allows you to retrieve the object corresponding to one of the screen layers. This object implements all the graphical routines.

A real example

Launch you Java environment and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-MaxiDisplay-G** 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.

UNABLE TO INCLUDE
http://172.17.17.77/tu/projects/yoctodisplay-128x64-G/public/examples/Java/helloworld.java

13.3. Control of the module part

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

```
import com.yoctopuce.YoctoAPI.*;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Demo {
```

```
    public static void main(String[] args)
    {
        try {
            // setup the API to use local VirtualHub
            YAPI.RegisterHub("127.0.0.1");
        } catch (YAPI_Exception ex) {
            System.out.println("Cannot contact VirtualHub on 127.0.0.1 (" +
ex.getLocalizedMessage() + ")");
            System.out.println("Ensure that the VirtualHub application is running");
            System.exit(1);
        }
        System.out.println("usage: demo [serial or logical name] [ON/OFF]");

        YModule module;
        if (args.length == 0) {
            module = YModule.FirstModule();
            if (module == null) {
                System.out.println("No module connected (check USB cable)");
                System.exit(1);
            }
        } else {
            module = YModule.FindModule(args[0]);  // use serial or logical name
        }

        try {
            if (args.length > 1) {
                if (args[1].equalsIgnoreCase("ON")) {
                    module.setBeacon(YModule.BEACON_ON);
                } else {
                    module.setBeacon(YModule.BEACON_OFF);
                }
            }
            System.out.println("serial:       " + module.get_serialNumber());
            System.out.println("logical name: " + module.get_logicalName());
            System.out.println("luminosity:   " + module.get_luminosity());
            if (module.get_beacon() == YModule.BEACON_ON) {
                System.out.println("beacon:       ON");
            } else {
                System.out.println("beacon:       OFF");
            }
            System.out.println("upTime:       " + module.get_upTime() / 1000 + " sec");
            System.out.println("USB current:  " + module.get_usbCurrent() + " mA");
            System.out.println("logs:\n" + module.get_lastLogs());
        } catch (YAPI_Exception ex) {
            System.out.println(args[1] + " not connected (check identification and USB
cable)");
        }
        YAPI.FreeAPI();
    }
}
```

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

Changing the module settings

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

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

public class Demo {

    public static void main(String[] args)
    {
        try {
            // setup the API to use local VirtualHub
            YAPI.RegisterHub("127.0.0.1");
```

```
        } catch (YAPI_Exception ex) {
            System.out.println("Cannot contact VirtualHub on 127.0.0.1 (" +
ex.getLocalizedMessage() + ")");
            System.out.println("Ensure that the VirtualHub application is running");
            System.exit(1);
        }

        if (args.length != 2) {
            System.out.println("usage: demo <serial or logical name> <new logical name>");
            System.exit(1);
        }

        YModule m;
        String newname;

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

        try {
            newname = args[1];
            if (!YAPI.CheckLogicalName(newname))
                {
                    System.out.println("Invalid name (" + newname + ")");
                    System.exit(1);
                }

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

            System.out.println("Module: serial= " + m.get_serialNumber());
            System.out.println(" / name= " + m.get_logicalName());
        } catch (YAPI_Exception ex) {
            System.out.println("Module " + args[0] + "not connected (check identification
and USB cable)");
            System.out.println(ex.getMessage());
            System.exit(1);
        }

        YAPI.FreeAPI();
    }
}
```

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

Listing the modules

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

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

public class Demo {

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

        System.out.println("Device list");
        YModule module = YModule.FirstModule();
        while (module != null) {
            try {
```

```
                System.out.println(module.get_serialNumber() + " (" +
module.get_productName() + ")");
            } catch (YAPI_Exception ex) {
                break;
            }
            module = module.nextModule();
        }
        YAPI.FreeAPI();
    }
}
```

13.4. Error handling

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

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

In the Java API, error handling is implemented with exceptions. Therefore you must catch and handle correctly all exceptions that might be thrown by the API if you do not want your software to crash as soon as you unplug a device.

14. Using the Yocto-MaxiDisplay-G with Android

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

14.1. Native access and VirtualHub

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

14.2. 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.

14.3. Compatibility

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

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

Android 4.x

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

USB *host* support

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

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

Supported hardware

The library is tested and validated on the following machines:

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

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

14.4. Activating the USB port under Android

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

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

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

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

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

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

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

Autorun

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

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

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

</manifest>
```

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

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

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

14.5. Control of the Display function

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

```
[...]
// Enable detection of USB devices
YAPI.EnableUSBHost(this);
YAPI.RegisterHub("usb");
[...]
// Retrieve the object used to interact with the device
display = YDisplay.FindDisplay("YD128G64-123456.display");

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

[...]
```

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.

YDisplay.FindDisplay

The `YDisplay.FindDisplay` function allows you to find a display 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-MaxiDisplay-G module with serial number *YD128G64-123456* which you have named "*MyModule*", and for which you have given the *display* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
display = YDisplay.FindDisplay("YD128G64-123456.display")
display = YDisplay.FindDisplay("YD128G64-123456.MyFunction")
display = YDisplay.FindDisplay("MyModule.display")
display = YDisplay.FindDisplay("MyModule.MyFunction")
display = YDisplay.FindDisplay("MyFunction")
```

`YDisplay.FindDisplay` returns an object which you can then use at will to control the display.

isOnline

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

get_displayLayer

The `get_displayLayer()` method of the object returned by `YDisplay.FindDisplay` allows you to retrieve the object corresponding to one of the screen layers. This object implements all the graphical routines.

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.

UNABLE TO INCLUDE http://172.17.17.77/tu/projects/yoctodisplay-128x64-G/public/examples/Android/helloworld.java

14.6. Control of the module part

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

```
package com.yoctopuce.doc_examples;

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

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

public class ModuleControl extends Activity implements OnItemSelectedListener
{

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

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

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

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

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

```
    }

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

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

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

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

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

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

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

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

Changing the module settings

When you want to modify the settings of a module, you only need to call the corresponding `YModule.set_xxx()` function. However, this modification is performed only in the random access memory (RAM) of the module: if the module is restarted, the modifications are lost. To memorize them persistently, it is necessary to ask the module to save its current configuration in its permanent

memory. To do so, use the `YModule.saveToFlash()` method. Inversely, it is possible to force the module to forget its current settings by using the `YModule.revertFromFlash()` method. The short example below allows you to modify the logical name of a module.

```
package com.yoctopuce.doc_examples;

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

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

public class SaveSettings extends Activity implements OnItemSelectedListener
{

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

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

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

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

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

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

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

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

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

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

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

}
```

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

Listing the modules

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

```
package com.yoctopuce.doc_examples;

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

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

public class Inventory extends Activity
{

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
```

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.inventory);
    }

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

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

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

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

}
```

14.7. Error handling

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

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

In the Java API for Android, error handling is implemented with exceptions. Therefore you must catch and handle correctly all exceptions that might be thrown by the API if you do not want your software to crash soon as you unplug a device.

15. Using Yocto-MaxiDisplay-G 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`.

15.1. Using the Yoctopuce library for TypeScript

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

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

```
npm install -g typescript
```

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

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

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

```
npm install
```

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

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

15.2. Refresher on asynchronous I/O in JavaScript

JavaScript is single-threaded by design. In order to handle time-consuming I/O operations, JavaScript relies on asynchronous operations: the I/O call is only triggered but then the code execution flow is suspended. The JavaScript engine is therefore free to handle other pending tasks, such as user interface. Whenever the pending I/O call is completed, the system invokes a callback function with the result of the I/O call to resume execution of the original execution flow.

When used with plain callback functions, as pervasive in Node.js libraries, asynchronous I/O tend to produce code with poor readability, as the execution flow is broken into many disconnected callback functions. Fortunately, the ECMAScript 2015 standard came in with *Promise* objects and a new `async` / `await` syntax to abstract calls to asynchronous calls:

- a function declared *async* automatically encapsulates its result as a Promise
- within an *async* function, any function call prefixed with 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.

15.3. Control of the Display function

A few lines of code are enough to use a Yocto-MaxiDisplay-G. Here is the skeleton of a TypeScript code snipplet to use the Display function.

```
// For Node.js, the library is referenced through the NPM package
// For HTML, we would use instead a relative path (depending on the build environment)
import { YAPI, YErrorMsg, YModule } from 'yoctolib-cjs/yocto_api_nodejs.js';
import { YDisplay } from 'yoctolib-cjs/yocto_display.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 display: YDisplay = YDisplay.FindDisplay("YD128G64-123456.display");

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

Let us look at these lines in more details.

yocto_api and yocto_display import

These two imports provide access to functions allowing you to manage Yoctopuce modules. `yocto_api` is always needed, `yocto_display` is necessary to manage modules containing a display, such as Yocto-MaxiDisplay-G. 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.

YDisplay.FindDisplay

The `FindDisplay` method allows you to find a display 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-MaxiDisplay-G module with serial number *YD128G64-123456* which you have named "*MyModule*", and for which you have given the *display* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
display = YDisplay.FindDisplay("YD128G64-123456.display")
display = YDisplay.FindDisplay("YD128G64-123456.MaFonction")
display = YDisplay.FindDisplay("MonModule.display")
display = YDisplay.FindDisplay("MonModule.MaFonction")
display = YDisplay.FindDisplay("MaFonction")
```

`YDisplay.FindDisplay` returns an object which you can then use at will to control the display.

isOnline

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

get_displayLayer

The `get_displayLayer()` method of the object returned by `YDisplay.FindDisplay` allows you to retrieve the object corresponding to one of the screen layers. This object implements all the graphical routines.

A real example, for Node.js

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

UNABLE TO INCLUDE
http://172.17.17.77/tu/projects/yoctodisplay-128x64-G/public/examples/typescript/helloworld.ts

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

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

```
npm run app-server
```

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

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

15.4. Control of the module part

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

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

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

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

    // Select the device to use
    let module: YModule = YModule.FindModule(args[0]);
    if(await module.isOnline()) {
        if(args.length > 1) {
            if(args[1] == 'ON') {
                await module.set_beacon(YModule.BEACON_ON);
            } else {
                await module.set_beacon(YModule.BEACON_OFF);
            }
        }
        console.log('serial:       '+await module.get_serialNumber());
```

```
        console.log('logical name: '+await module.get_logicalName());
        console.log('luminosity:   '+await module.get_luminosity()+'%');
        console.log('beacon:       '+
            (await module.get_beacon() == YModule.BEACON_ON ? 'ON' : 'OFF'));
        console.log('upTime:       '+
            ((await module.get_upTime()/1000)>>0) +' sec');
        console.log('USB current:  '+await module.get_usbCurrent()+' mA');
        console.log('logs:');
        console.log(await module.get_lastLogs());
    } else {
        console.log("Module not connected (check identification and USB cable)\n");
    }
    await YAPI.FreeAPI();
}

if(process.argv.length < 3) {
    console.log("usage: npm run demo <serial or logicalname> [ ON | OFF ]");
} else {
    startDemo(process.argv.slice(2));
}
```

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

Changing the module settings

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

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

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

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

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

if(process.argv.length < 3) {
    console.log("usage: npm run demo <serial> [newLogicalName]");
} else {
    startDemo(process.argv.slice(2));
}
```

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

Listing the modules

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

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

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

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

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

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

startDemo();
```

15.5. Error handling

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

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

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

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

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

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

16. Using Yocto-MaxiDisplay-G with JavaScript / EcmaScript

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

16.2. Using Yoctopuce library for JavaScript / EcmaScript 2017

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

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

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

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

Using the official Yoctopuce library for node.js

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

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

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

```
npm install
```

Once done, you can start the example file using:

```
node demo.js
```

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

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

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

Using the Yoctopuce library within a browser (HTML)

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

Using the Yoctoluce library on older JavaScript engines

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

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

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

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

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

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

Backward-compatibility with the old JavaScript library

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

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

by

[3] http://babeljs.io

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

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

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

by

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

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

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

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

you can use:

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

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

You can also rewrite this last asynchronous call as:

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

16.3. Control of the Display function

A few lines of code are enough to use a Yocto-MaxiDisplay-G. Here is the skeleton of a JavaScript code snippet to use the Display 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_display.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 display = YDisplay.FindDisplay("YD128G64-123456.display");

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

```
if(await display.isOnline())
{
    // Use display.get_displayLayer()
    [...]
}
```

Let us look at these lines in more details.

yocto_api and yocto_display import

These two import provide access to functions allowing you to manage Yoctopuce modules. `yocto_api` is always needed, `yocto_display` is necessary to manage modules containing a display, such as Yocto-MaxiDisplay-G. 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.

YDisplay.FindDisplay

The `FindDisplay` method allows you to find a display 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-MaxiDisplay-G module with serial number *YD128G64-123456* which you have named "*MyModule*", and for which you have given the *display* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
display = YDisplay.FindDisplay("YD128G64-123456.display")
display = YDisplay.FindDisplay("YD128G64-123456.MaFonction")
display = YDisplay.FindDisplay("MonModule.display")
display = YDisplay.FindDisplay("MonModule.MaFonction")
display = YDisplay.FindDisplay("MaFonction")
```

`YDisplay.FindDisplay` returns an object which you can then use at will to control the display.

isOnline

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

get_displayLayer

The `get_displayLayer()` method of the object returned by `YDisplay.FindDisplay` allows you to retrieve the object corresponding to one of the screen layers. This object implements all the graphical routines.

A real example, for Node.js

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

UNABLE TO INCLUDE
http://172.17.17.77/tu/projects/yoctodisplay-128x64-G/public/examples/ecmascript/node.js

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-MaxiDisplay-G**. 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.

UNABLE TO INCLUDE
http://172.17.17.77/tu/projects/yoctodisplay-128x64-G/public/examples/ecmascript/helloworld.html

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

16.4. Control of the module part

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

```
"use strict";

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

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

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

    // Select the relay to use
    let module = YModule.FindModule(args[0]);
    if(await module.isOnline()) {
        if(args.length > 1) {
            if(args[1] == 'ON') {
                await module.set_beacon(YModule.BEACON_ON);
            } else {
                await module.set_beacon(YModule.BEACON_OFF);
            }
        }
        console.log('serial:       '+await module.get_serialNumber());
        console.log('logical name: '+await module.get_logicalName());
        console.log('luminosity:   '+await module.get_luminosity()+'%');
        console.log('beacon:       '+(await module.get_beacon()==YModule.BEACON_ON
?'ON':'OFF'));
        console.log('upTime:       '+parseInt(await module.get_upTime()/1000)+' sec');
        console.log('USB current:  '+await module.get_usbCurrent()+' mA');
        console.log('logs:');
```

```
        console.log(await module.get_lastLogs());
    } else {
        console.log("Module not connected (check identification and USB cable)\n");
    }
    await YAPI.FreeAPI();
}

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

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

Changing the module settings

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

```
"use strict";

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

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

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

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

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

Warning: the number of write cycles of the nonvolatile memory of the module is limited. When this limit is reached, nothing guaranties that the saving process is performed correctly. This limit, linked to the technology employed by the module micro-processor, is located at about 100000 cycles. In short,

you can use the `saveToFlash()` function only 100000 times in the life of the module. Make sure you do not call this function within a loop.

Listing the modules

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

```
"use strict";

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

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

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

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

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

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

16.5. Error handling

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

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

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

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

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

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

17. Using Yocto-MaxiDisplay-G with PHP

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

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

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

17.1. Getting ready

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

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

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

17.2. Control of the Display function

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

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

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

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

// Retrieve the object used to interact with the device
$display = YDisplay::FindDisplay("YD128G64-123456.display");

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

Let's look at these lines in more details.

yocto_api.php and yocto_display.php

These two PHP includes provides access to the functions allowing you to manage Yoctopuce modules. `yocto_api.php` must always be included, `yocto_display.php` is necessary to manage modules containing a display, such as Yocto-MaxiDisplay-G.

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.

YDisplay::FindDisplay

The `YDisplay::FindDisplay` function allows you to find a display 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-MaxiDisplay-G module with serial number *YD128G64-123456* which you have named "*MyModule*", and for which you have given the *display* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
$display = YDisplay::FindDisplay("YD128G64-123456.display");
$display = YDisplay::FindDisplay("YD128G64-123456.MyFunction");
$display = YDisplay::FindDisplay("MyModule.display");
$display = YDisplay::FindDisplay("MyModule.MyFunction");
$display = YDisplay::FindDisplay("MyFunction");
```

`YDisplay::FindDisplay` returns an object which you can then use at will to control the display.

isOnline

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

get_displayLayer

The `get_diplayLayer()` method of the object returned by `YFindDisplay` allows you to retrieve the object corresponding to one of the screen layers. This object implements all the graphical routines.

A real example

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

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

this page. The code is also provided in the directory **Examples/Doc-GettingStarted-Yocto-MaxiDisplay-G** 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.

UNABLE TO INCLUDE
http://172.17.17.77/tu/projects/yoctodisplay-128x64-G/public/examples/php/helloworld.php

17.3. Control of the module part

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

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

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

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

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

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

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

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

Changing the module settings

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

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

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

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

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

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

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

Listing the modules

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

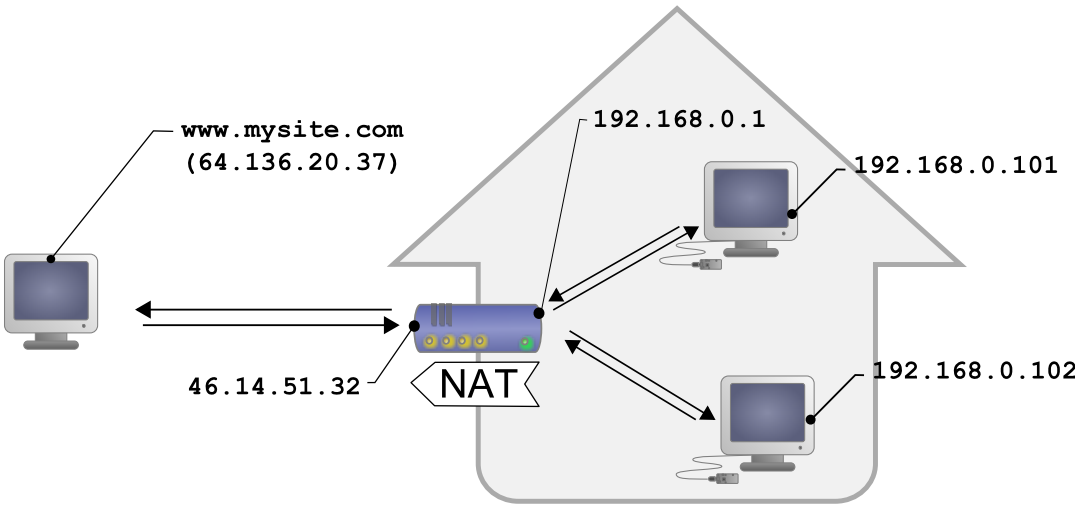
```
<HTML>
<HEAD>
 <TITLE>inventory</TITLE>
</HEAD>
<BODY>
<H1>Device list</H1>
<TT>
<?php
    include('yocto_api.php');
    YAPI::RegisterHub("http://127.0.0.1:4444/");
    $module   = YModule::FirstModule();
    while (!is_null($module)) {
        printf("%s (%s)<br>", $module->get_serialNumber(),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

17.4. HTTP callback API and NAT filters

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

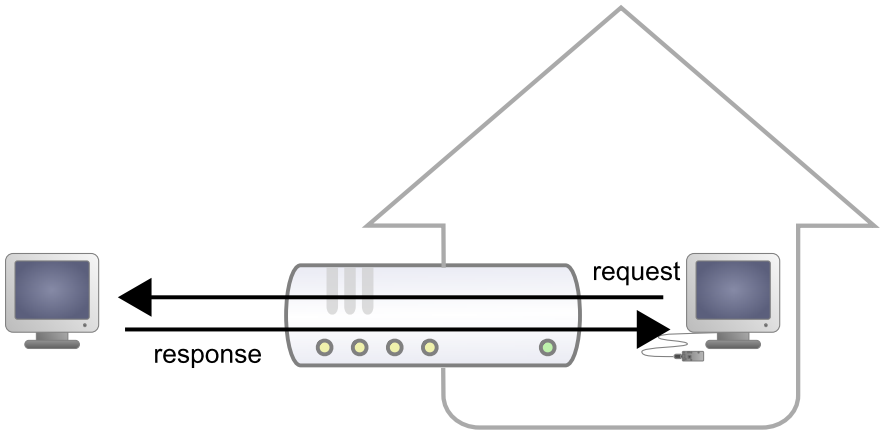
The NAT filter: advantages and disadvantages

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

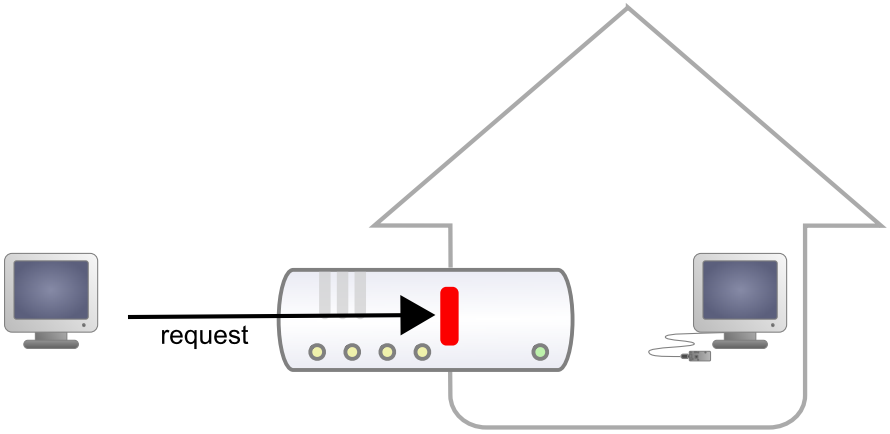


Typical DSL configuration: LAN machines are isolated from the outside by the DSL router

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



Responses from request from LAN machines are routed.

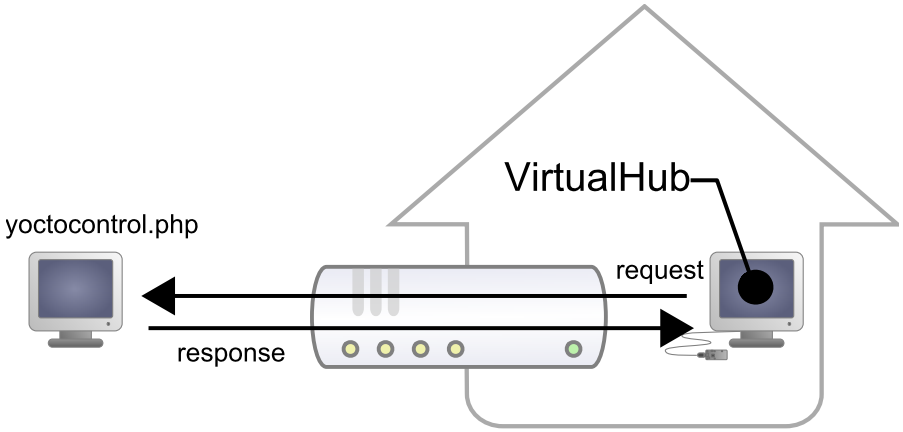


But requests from the outside are blocked.

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

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

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



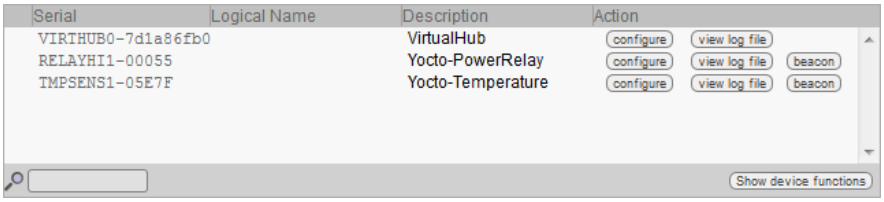
The HTTP callback API uses the VirtualHub which initiates the requests.

Configuration

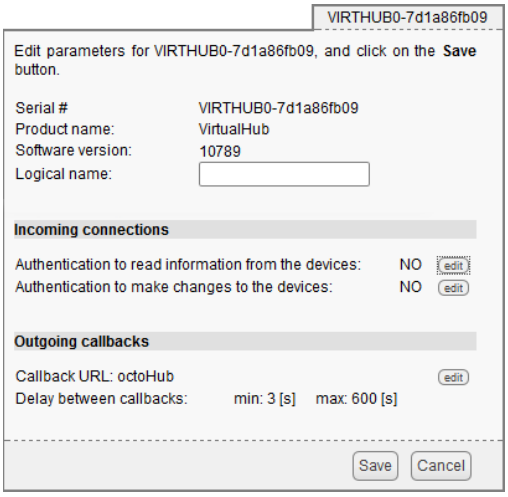
The callback API thus uses the *VirtualHub* as a gateway. All the communications are initiated by the *VirtualHub*. They are thus outgoing communications and therefore perfectly authorized by the DSL router.

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

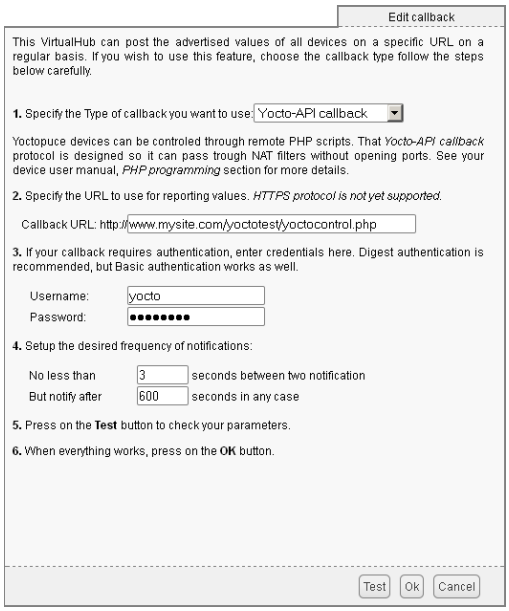
1. Launch a *VirtualHub*
2. Access its interface, usually 127.0.0.1:4444
3. Click on the **configure** button of the line corresponding to the *VirtualHub* itself
4. Click on the **edit** button of the **Outgoing callbacks** section



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.

17.5. Error handling

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

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

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

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

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

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

18. Using Yocto-MaxiDisplay-G with Visual Basic .NET

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

18.1. Installation

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

18.2. Using the Yoctopuce API in a Visual Basic project

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

Configuring a Visual Basic project

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

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

You then have the choice between simply adding these files to your project, or to add them as links (the **Add** button is in fact a scroll-down menu). In the first case, Visual Studio copies the selected

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

files into your project. In the second case, Visual Studio simply keeps a link on the original files. We recommend you to use links, which makes updates of the library much easier.

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

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

18.3. Control of the Display function

A few lines of code are enough to use a Yocto-MaxiDisplay-G. Here is the skeleton of a Visual Basic code snipplet to use the Display 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 display As YDisplay
display = YDisplay.FindDisplay("YD128G64-123456.display")

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

YDisplay.FindDisplay

The `YDisplay.FindDisplay` function allows you to find a display 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-MaxiDisplay-G module with serial number *YD128G64-123456* which you have named "*MyModule*", and for which you have given the *display* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
display = YDisplay.FindDisplay("YD128G64-123456.display")
display = YDisplay.FindDisplay("YD128G64-123456.MyFunction")
display = YDisplay.FindDisplay("MyModule.display")
display = YDisplay.FindDisplay("MyModule.MyFunction")
display = YDisplay.FindDisplay("MyFunction")
```

`YDisplay.FindDisplay` returns an object which you can then use at will to control the display.

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

isOnline

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

get_displayLayer

The `get_diplayLayer()` method of the object returned by `YFindDisplay` allows you to retrieve the object corresponding to one of the screen layers. This object implements all the graphical routines.

A real example

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

UNABLE TO INCLUDE
http://172.17.17.77/tu/projects/yoctodisplay-128x64-G/public/examples/VB/helloworld.vb

18.4. Control of the module part

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

```
Imports System.IO
Imports System.Environment

Module Module1

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

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

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

    If argv.Length < 2 Then usage()

    m = YModule.FindModule(argv(1)) REM use serial or logical name
    If (m.isOnline()) Then
      If argv.Length > 2 Then
        If argv(2) = "ON" Then m.set_beacon(Y_BEACON_ON)
        If argv(2) = "OFF" Then m.set_beacon(Y_BEACON_OFF)
      End If
      Console.WriteLine("serial:       " + m.get_serialNumber())
      Console.WriteLine("logical name: " + m.get_logicalName())
      Console.WriteLine("luminosity:   " + Str(m.get_luminosity()))
      Console.Write("beacon:       ")
      If (m.get_beacon() = Y_BEACON_ON) Then
        Console.WriteLine("ON")
      Else
        Console.WriteLine("OFF")
      End If
      Console.WriteLine("upTime:       " + Str(m.get_upTime() / 1000) + " sec")
      Console.WriteLine("USB current:  " + Str(m.get_usbCurrent()) + " mA")
      Console.WriteLine("Logs:")
      Console.WriteLine(m.get_lastLogs())
    Else
```

```
      Console.WriteLine(argv(1) + " not connected (check identification and USB cable)")
    End If
    YAPI.FreeAPI()
  End Sub

End Module
```

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

Changing the module settings

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

```
Module Module1


  Sub usage()

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

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

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

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

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

  End Sub

End Module
```

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

Listing the modules

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

```
Module Module1

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

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

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

End Module
```

18.5. Error handling

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

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

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

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

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

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

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

19. Using Yocto-MaxiDisplay-G with Delphi

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

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

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

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

19.1. Preparation

Go to the Yoctopuce web site and download the Yoctopuce Delphi libraries[3]. Uncompress everything in a directory of your choice, add the subdirectory *sources* in the list of directories of Delphi libraries.[4]

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

19.2. Control of the Display function

A few lines of code are enough to use a Yocto-MaxiDisplay-G. Here is the skeleton of a Delphi code snipplet to use the Display function.

```
uses yocto_api, yocto_display;

var errmsg: string;
    display: TYDisplay;

[...]
```

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

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

// Retrieve the object used to interact with the device
display = yFindDisplay("YD128G64-123456.display")

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

Let's look at these lines in more details.

yocto_api and yocto_display

These two units provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api` must always be used, `yocto_display` is necessary to manage modules containing a display, such as Yocto-MaxiDisplay-G.

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.

yFindDisplay

The `yFindDisplay` function allows you to find a display 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-MaxiDisplay-G module with serial number *YD128G64-123456* which you have named "*MyModule*", and for which you have given the *display* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
display := yFindDisplay("YD128G64-123456.display");
display := yFindDisplay("YD128G64-123456.MyFunction");
display := yFindDisplay("MyModule.display");
display := yFindDisplay("MyModule.MyFunction");
display := yFindDisplay("MyFunction");
```

`yFindDisplay` returns an object which you can then use at will to control the display.

isOnline

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

get_displayLayer

The `get_diplayLayer()` method of the object returned by `YFindDisplay` allows you to retrieve the object corresponding to one of the screen layers. This object implements all the graphical routines.

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.

UNABLE TO INCLUDE
http://172.17.17.77/tu/projects/yoctodisplay-128x64-G/public/examples/delphi/helloworld.dpr

19.3. Control of the module part

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

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

const
  serial = 'YD128G64-123456'; // 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 = 'YD128G64-123456'; // 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.
```

19.4. Error handling

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

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

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

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

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

information, the return value is YAPI_SUCCESS if everything went well, and a different error code in case of failure.

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

20. Using the Yocto-MaxiDisplay-G 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.

20.1. Blocking and asynchronous functions

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

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

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

Example:

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

    return result;
}

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

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

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

For you not to have to wonder 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.

20.2. 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].

20.3. Using the Yoctopuce API in a Visual Studio project

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

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

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

The Package.appxmanifest file

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

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

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

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

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

```
<DeviceCapability Name="humaninterfacedevice">
    <!-- Yocto-MaxiDisplay-G -->
    <Device Id="vidpid:24e0 004A">
      <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.

20.4. Control of the Display function

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

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

// Retrieve the object used to interact with the device
YDisplay display = YDisplay.FindDisplay("YD128G64-123456.display");

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

[...]
```

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.

YDisplay.FindDisplay

The `YDisplay.FindDisplay` function allows you to find a display 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-MaxiDisplay-G module with serial number *YD128G64-123456* which you have named "*MyModule*", and for which you have given the *display* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
display = YDisplay.FindDisplay("YD128G64-123456.display");
display = YDisplay.FindDisplay("YD128G64-123456.MaFonction");
display = YDisplay.FindDisplay("MonModule.display");
display = YDisplay.FindDisplay("MonModule.MaFonction");
display = YDisplay.FindDisplay("MaFonction");
```

`YDisplay.FindDisplay` returns an object which you can then use at will to control the display.

isOnline

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

get_displayLayer

The `get_displayLayer()` method of the object returned by `YDisplay.FindDisplay` allows you to retrieve the object corresponding to one of the screen layers. This object implements all the graphical routines.

20.5. A real example

Launch Visual Studio and open the corresponding project provided in the directory **Examples/Doc-GettingStarted-Yocto-MaxiDisplay-G** 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.

UNABLE TO INCLUDE
http://172.17.17.77/tu/projects/yoctodisplay-128x64-G/public/examples/UWP/helloworld.cs

20.6. Control of the module part

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

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

namespace Demo
{
  public class Demo : DemoBase
  {

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

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

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

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

```
      return 0;
    }
  }
}
```

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

Changing the module settings

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

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

namespace Demo
{
  public class Demo : DemoBase
  {

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

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

        await YAPI.RegisterHub(HubURL);

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

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

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

Listing the modules

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

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

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

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

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

20.7. Error handling

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

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

In the Universal Windows Platform library, error handling is implemented with exceptions. You must therefore intercept and correctly handle these exceptions if you want to have a reliable project which does not crash as soon as you disconnect a module.

Library thrown exceptions are always of the YAPI_Exception type, so you can easily separate them from other exceptions in a `try{...} catch{...}` block.

Example:

```
try {
        ....
```

```
} catch (YAPI_Exception ex) {
        Debug.WriteLine("Exception from Yoctopuce lib:" + ex.Message);
} catch (Exception ex) {
        Debug.WriteLine("Other exceptions :" + ex.Message);
}
```

21. Using Yocto-MaxiDisplay-G with Objective-C

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

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

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

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

21.1. Control of the Display function

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

```
#import "yocto_api.h"
#import "yocto_display.h"

...
NSError *error;
[YAPI RegisterHub:@"usb": &error]
...
// On récupère l'objet représentant le module (ici connecté en local sur USB)
display = [YDisplay FindDisplay:@"YD128G64-123456.display"];

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

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

```
    // Utiliser [display get_displayLayer]
    ...
}
```

Let's look at these lines in more details.

yocto_api.h and yocto_display.h

These two import files provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api.h` must always be used, `yocto_display.h` is necessary to manage modules containing a display, such as Yocto-MaxiDisplay-G.

[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.

[Display FindDisplay]

The `[Display FindDisplay]` function allows you to find a display 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-MaxiDisplay-G module with serial number *YD128G64-123456* which you have named "*MyModule*", and for which you have given the *display* function the name "*MyFunction*". The following five calls are strictly equivalent, as long as "*MyFunction*" is defined only once.

```
YDisplay *display = [Display FindDisplay:@"YD128G64-123456.display"];
YDisplay *display = [Display FindDisplay:@"YD128G64-123456.MyFunction"];
YDisplay *display = [Display FindDisplay:@"MyModule.display"];
YDisplay *display = [Display FindDisplay:@"MyModule.MyFunction"];
YDisplay *display = [Display FindDisplay:@"MyFunction"];
```

`[Display FindDisplay]` returns an object which you can then use at will to control the display.

isOnline

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

get_displayLayer

The `get_displayLayer()` method of the object returned by `YDisplay.FindDisplay` allows you to retrieve the object corresponding to one of the screen layers. This object implements all the graphical routines.

A real example

Launch Xcode 4.2 and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-MaxiDisplay-G** 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.

UNABLE TO INCLUDE http://172.17.17.77/tu/projects/yoctodisplay-128x64-G/public/examples/Objective-C/helloworld.m

21.2. Control of the module part

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

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

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


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

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

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

Changing the module settings

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

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

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

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

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

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

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

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

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

Listing the modules

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

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

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

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

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

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

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

21.3. Error handling

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

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

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

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

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

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

22. Using with unsupported languages

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

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

22.1. Command line

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

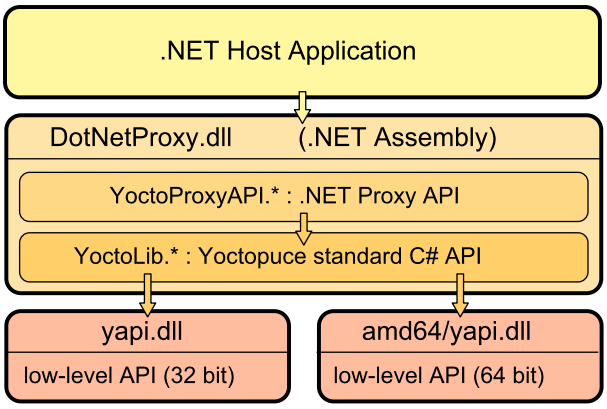
However, if the command line API is the easiest solution, it is neither the fastest nor the most efficient. For each call, the executable must initialize its own API and make an inventory of USB connected modules. This requires about one second per call.

22.2. .NET Assembly

A .NET Assembly enables you to share a set of pre-compiled classes to offer a service, by stating entry points which can be used by third-party applications. In our case, it's the whole Yoctopuce library which is available in the .NET Assembly, so that it can be used in any environment which supports .NET Assembly dynamic loading.

The Yoctopuce library as a .NET Assembly does not contain only the standard C# Yoctopuce library, as this wouldn't have allowed an optimal use in all environments. Indeed, we cannot expect host applications to necessarily offer a thread system or a callback system, although they are very useful to manage plug-and-play events and sensors with a high refresh rate. Likewise, we can't expect from external applications a transparent behavior in cases where a function call in Assembly creates a delay because of network communications.

Therefore, we added to it an additional layer, called *.NET Proxy* library. This additional layer offers an interface very similar to the standard library but somewhat simplified, as it internally manages all the callback mechanisms. Instead, this library offers mirror objects, called *Proxys*, which publish through *Properties* the main attributes of the Yoctopuce functions such as the current measure, configuration parameters, the state, and so on.



.NET Assembly Architecture

The callback mechanism automatically updates the properties of the *Proxys* objects, without the host application needing to care for it. The later can thus, at any time and without any risk of latency, display the value of all properties of Yoctopuce Proxy objects.

Pay attention to the fact that the `yapi.dll` low-level communication library is **not** included in the .NET Assembly. You must therefore keep it together with `DotNetProxyLibrary.dll`. The 32 bit version must be located in the same directory as `DotNetProxyLibrary.dll`, while the 64 bit version must be in a subdirectory `amd64`.

Example of use with MATLAB

Here is how to load our Proxy .NET Assembly in MATLAB and how to read the value of the first sensor connected by USB found on the machine:

```
NET.addAssembly("C:/Yoctopuce/DotNetProxyLibrary.dll");
import YoctoProxyAPI.*

errmsg = YAPIProxy.RegisterHub("usb");
sensor = YSensorProxy.FindSensor("");
measure = sprintf('%.3f %s', sensor.CurrentValue, sensor.Unit);
```

Example of use in PowerShell

PowerShell commands are a little stranger, but we can recognize the same structure:

```
Add-Type -Path "C:/Yoctopuce/DotNetProxyLibrary.dll"

$errmsg = [YoctoProxyAPI.YAPIProxy]::RegisterHub("usb")
$sensor = [YoctoProxyAPI.YSensorProxy]::FindSensor("")
$measure = "{0:n3} {1}" -f $sensor.CurrentValue, $sensor.Unit
```

Specificities of the .NET Proxy library

With regards to classic Yoctopuce libraries, the following differences in particular should be noted:

No FirstModule/nextModule method

To obtain an object referring to the first found module, we call `YModuleProxy.FindModule("")`. If there is no connected module, this method returns an object with its `module.IsOnline` property set to `False`. As soon as a module is connected, the property changes to `True` and the module hardware identifier is updated.

To list modules, you can call the `module.GetSimilarFunctions()` method which returns an array of character strings containing the identifiers of all the modules which were found.

No callback function

Callback functions are implemented internally and they update the object properties. You can therefore simply poll the properties, without significant performance penalties. Be aware that if you

use one of the function that disables callbacks, the automatic refresh of object properties may not work anymore.

A new method `YAPIProxy.GetLog` makes it possible to retrieve low-level debug logs without using callbacks.

Enumerated types
In order to maximize compatibility with host applications, the .NET Proxy library does not use true .NET enumerated types, but simple integers. For each enumerated type, the library includes public constants named according to the possible values. Contrarily to standard Yoctopuce libraries, numeric values always start from `1`, as the value `0` is reserved to return an invalid value, for instance when the device is disconnected.

Invalid numeric results
For all numeric results, rather than using an arbitrary constant, the invalid value returned in case of error is *NaN*. You should therefore use function *isNaN()* to detect this value.

Using .NET Assembly without the Proxy library

If for a reason or another you don't want to use the Proxy library, and if your environment allows it, you can use the standard C# API as it is located in the Assembly, under the `YoctoLib` namespace. Beware however not to mix both types of use: either you go through the Proxy library, or you use he `YoctoLib` version directly, but not both!

CompatibilitÃ©

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

In order to be compatible with as many versions of Windows as possible, including Windows XP, the *DotNetProxyLibrary.dll* library is compiled in .NET 3.5, which is available by default on all the Windows versions since XP. As of today, we have never met any non-Windows environment able to load a .NET Assembly, so we only ship the low-level communication dll for Windows together with the assembly.

22.3. VirtualHub and HTTP GET

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

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

REST interface

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

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

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

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

Driving a module through the REST interface

Each Yoctopuce module has its own REST interface, available in several variants. Let us imagine a Yocto-MaxiDisplay-G with the *YD128G64-12345* 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/YD128G64-12345/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/YD128G64-12345/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/YD128G64-12345/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 display function, build the following URL:

```
http://127.0.0.1:4444/bySerial/YD128G64-12345/api/display.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/YD128G64-12345/api/display/logicalName
```

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

```
http://127.0.0.1:4444/bySerial/YD128G64-12345/api/display?logicalName=myFunction
```

You can find the list of available attributes for your Yocto-MaxiDisplay-G 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/YD128G64-12345/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/YD128G64-12345/dataLogger.json?id=display&utc=1389801080
```

22.4. Using dynamic libraries

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

Filename	Platform
libyapi.dylib	Max OS X
libyapi-amd64.so	Linux Intel (64 bits)
libyapi-armel.so	Linux ARM EL (32 bits)
libyapi-armhf.so	Linux ARM HL (32 bits)
libyapi-aarch64.so	Linux ARM (64 bits)
libyapi-i386.so	Linux Intel (32 bits)
yapi64.dll	Windows (64 bits)
yapi.dll	Windows (32 bits)

These dynamic libraries contain all the functions necessary to completely rebuild the whole high level API in any language able to integrate these libraries. This chapter nevertheless restrains itself to describing basic use of the modules.

Driving a module

The three essential functions of the low level API are the following:

```
int yapiInitAPI(int connection_type, char *errmsg);
int yapiUpdateDeviceList(int forceupdate, char *errmsg);
int yapiHTTPRequest(char *device, char *request, char* buffer,int buffsize,int *fullsize,
char *errmsg);
```

The *yapiInitAPI* function initializes the API and must be called once at the beginning of the program. For a USB type connection, the *connection_type* parameter takes value 1. The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message. This pointer can also point to *null*. The function returns a negative integer in case of error, zero otherwise.

The *yapiUpdateDeviceList* manages the inventory of connected Yoctopuce modules. It must be called at least once. To manage hot plug and detect potential newly connected modules, this function must be called at regular intervals. The *forceupdate* parameter must take value 1 to force a hardware scan. The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message. This pointer can also point to *null*. The function returns a negative integer in case of error, zero otherwise.

Finally, the *yapiHTTPRequest* function sends HTTP requests to the module REST API. The *device* parameter contains the serial number or the logical name of the module which you want to reach. The *request* parameter contains the full HTTP request (including terminal line breaks). *buffer* points to a character buffer long enough to contain the answer. *buffsize* is the size of the buffer. *fullsize* is a pointer to an integer to which will be assigned the actual size of the answer. The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message. This pointer can also point to *null*. The function returns a negative integer in case of error, zero otherwise.

The format of the requests is the same as the one described in the *VirtualHub et HTTP GET* section. All the character strings used by the API are strings made of 8-bit characters: Unicode and UTF8 are not supported.

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

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

```
// Dll functions import
function  yapiInitAPI(mode:integer;
                      errmsg : pansichar):integer;cdecl;
```

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

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

const
  serial      = 'YD128G64-12345';
  getValue = 'GET /api/module/luminosity HTTP/1.1'#13#10#13#10;
  setValue = 'GET /api/module?luminosity=100 HTTP/1.1'#13#10#13#10;

begin
  errmsg  :=  @errmsgBuffer;
  data    :=  @dataBuffer;
  // API  initialization
  if(yapiInitAPI(1,errmsg)<0) then
   begin
    writeln(errmsg);
    halt;
  end;

  // forces a device inventory
  if( yapiUpdateDeviceList(1,errmsg)<0) then
    begin
     writeln(errmsg);
     halt;
   end;

  // requests the  module luminosity
  if (yapiHTTPRequest(serial,getValue,data,sizeof(dataBuffer),fullsize,errmsg)<0) then
   begin
     writeln(errmsg);
     halt;
   end;

  // searches for the HTTP header end
  p := pos(#13#10#13#10,data);

  // displays the response minus the HTTP header
  writeln(copy(data,p+4,length(data)-p-3));

  // changes the luminosity
  if (yapiHTTPRequest(serial,setValue,data,sizeof(dataBuffer),fullsize,errmsg)<0) then
   begin
     writeln(errmsg);
     halt;
   end;

end.
```

Module inventory

To perform an inventory of Yoctopuce modules, you need two functions from the dynamic library:

```
int yapiGetAllDevices(int *buffer,int maxsize,int *neededsize,char *errmsg);
int yapiGetDeviceInfo(int devdesc,yDeviceSt *infos, char *errmsg);
```

The *yapiGetAllDevices* function retrieves the list of all connected modules as a list of handles. *buffer* points to a 32-bit integer array which contains the returned handles. *maxsize* is the size in bytes of the buffer. To *neededsize* is assigned the necessary size to store all the handles. From this, you can deduce either the number of connected modules or that the input buffer is too small. The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message. This pointer can also point to *null*. The function returns a negative integer in case of error, zero otherwise.

The *yapiGetDeviceInfo* function retrieves the information related to a module from its handle. *devdesc* is a 32-bit integer representing the module and which was obtained through *yapiGetAllDevices*. *infos* points to a data structure in which the result is stored. This data structure has the following format:

Name	Type	Size (bytes)	Description
vendorid	int	4	Yoctopuce USB ID
deviceid	int	4	Module USB ID
devrelease	int	4	Module version
nbinbterfaces	int	4	Number of USB interfaces used by the module
manufacturer	char[]	20	Yoctopuce (null terminated)
productname	char[]	28	Model (null terminated)
serial	char[]	20	Serial number (null terminated)
logicalname	char[]	20	Logical name (null terminated)
firmware	char[]	22	Firmware version (null terminated)
beacon	byte	1	Beacon state (0/1)

The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message.

Here is a sample program written in pascal using the *yapi.dll* DLL to list the connected modules.

```
// device description structure
type yDeviceSt = packed record
   vendorid        : word;
   deviceid        : word;
   devrelease      : word;
   nbinbterfaces   : word;
   manufacturer    : array [0..19] of ansichar;
   productname     : array [0..27] of ansichar;
   serial          : array [0..19] of ansichar;
   logicalname     : array [0..19] of ansichar;
   firmware        : array [0..21] of ansichar;
   beacon          : byte;
 end;

// Dll function import
function  yapiInitAPI(mode:integer;
                      errmsg : pansichar):integer;cdecl;
                      external 'yapi.dll' name 'yapiInitAPI';

function  yapiUpdateDeviceList(force:integer;errmsg : pansichar):integer;cdecl;
                      external 'yapi.dll' name 'yapiUpdateDeviceList';

function  yapiGetAllDevices( buffer:pointer;
                             maxsize:integer;
                             var neededsize:integer;
                             errmsg : pansichar):integer; cdecl;
                             external 'yapi.dll' name 'yapiGetAllDevices';

function  apiGetDeviceInfo(d:integer; var infos:yDeviceSt;
                             errmsg : pansichar):integer;  cdecl;
                             external 'yapi.dll' name 'yapiGetDeviceInfo';


var
 errmsgBuffer  : array [0..256] of ansichar;
 dataBuffer    : array [0..127] of integer;   // max of 128 USB devices
 errmsg,data   : pansichar;
 neededsize,i  : integer;
 devinfos      : yDeviceSt;

begin
  errmsg  :=  @errmsgBuffer;

  // API  initialization
  if(yapiInitAPI(1,errmsg)<0) then
   begin
    writeln(errmsg);
    halt;
  end;
```

```
   // forces a device inventory
  if( yapiUpdateDeviceList(1,errmsg)<0) then
    begin
     writeln(errmsg);
     halt;
   end;

  // loads all device handles into dataBuffer
  if yapiGetAllDevices(@dataBuffer,sizeof(dataBuffer),neededsize,errmsg)<0 then
    begin
     writeln(errmsg);
     halt;
    end;

  // gets device info from each handle
  for i:=0 to  neededsize div sizeof(integer)-1 do
   begin
     if (apiGetDeviceInfo(dataBuffer[i], devinfos, errmsg)<0) then
       begin
         writeln(errmsg);
         halt;
       end;
     writeln(pansichar(@devinfos.serial)+' ('+pansichar(@devinfos.productname)+')');
   end;

end.
```

VB6 and yapi.dll

Each entry point from the yapi.dll is duplicated. You will find one regular C-decl version and one Visual Basic 6 compatible version, prefixed with *vb6_*.

22.5. Porting the high level library

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

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

23. Advanced programming

The preceding chapters have introduced, in each available language, the basic programming functions which can be used with your Yocto-MaxiDisplay-G module. This chapter presents in a more generic manner a more advanced use of your module. Examples are provided in the language which is the most popular among Yoctopuce customers, that is C#. Nevertheless, you can find complete examples illustrating the concepts presented here in the programming libraries of each language.

To remain as concise as possible, examples provided in this chapter do not perform any error handling. Do not copy them "as is" in a production application.

23.1. Event programming

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

Detecting module arrival and departure

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

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

The callback

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

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

Initialization

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

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

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

Triggering callbacks

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

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

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

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

24. High-level API Reference

This chapter summarizes the high-level API functions to drive your Yocto-MaxiDisplay-G. Syntax and exact type names may vary from one language to another, but, unless otherwise stated, all the functions are available in every language. For detailed information regarding the types of arguments and return values for a given language, refer to the definition file for this language (`yocto_api.*` as well as the other `yocto_*` files that define the function interfaces).

For languages which support exceptions, all of these functions throw exceptions in case of error by default, rather than returning the documented error value for each function. This is by design, to facilitate debugging. It is however possible to disable the use of exceptions using the `yDisableExceptions()` function, in case you prefer to work with functions that return error values.

This chapter does not repeat the programming concepts described earlier, in order to stay as concise as possible. In case of doubt, do not hesitate to go back to the chapter describing in details all configurable attributes.

24.1. Class YAPI

General functions

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

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

java	import com.yoctopuce.YoctoAPI.YAPI;
dnp	import YoctoProxyAPI.YAPIProxy
cp	#include "yocto_api_proxy.h"
ml	import YoctoProxyAPI.YAPIProxy"
js	<script type='text/javascript' src='yocto_api.js'></script>
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
uwp	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *
php	require_once('yocto_api.php');
ts	in HTML: import { YAPI, YErrorMsg, YModule, YSensor } from '../../dist/esm/yocto_api_browser.js'; in Node.js: import { YAPI, YErrorMsg, YModule, YSensor } from 'yoctolib-cjs/yocto_api_nodejs.js';
es	in HTML: <script src="../../lib/yocto_api.js"></script> in node.js: require('yoctolib-es2017/yocto_api.js');
vi	YModule.vi

Global functions
YAPI.CheckLogicalName(**name**) Checks if a given string is valid as logical name for a module or a function.
YAPI.ClearHTTPCallbackCacheDir(**bool_removeFiles**) Disables the HTTP callback cache.
YAPI.DisableExceptions() Disables the use of exceptions to report runtime errors.
YAPI.EnableExceptions() Re-enables the use of exceptions for runtime error handling.
YAPI.EnableUSBHost(**osContext**) This function is used only on Android.
YAPI.FreeAPI() Waits for all pending communications with Yoctopuce devices to be completed then frees dynamically allocated resources used by the Yoctopuce library.
YAPI.GetAPIVersion() Returns the version identifier for the Yoctopuce library in use.
YAPI.GetCacheValidity() Returns the validity period of the data loaded by the library.
YAPI.GetDeviceListValidity()

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

YAPI.GetDllArchitecture()

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

YAPI.GetDllPath()

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

YAPI.GetLog(**lastLogLine**)

Retrieves Yoctopuce low-level library diagnostic logs.

YAPI.GetNetworkTimeout()

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

YAPI.GetTickCount()

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

YAPI.HandleEvents(**errmsg**)

Maintains the device-to-library communication channel.

YAPI.InitAPI(**mode**, **errmsg**)

Initializes the Yoctopuce programming library explicitly.

YAPI.PreregisterHub(**url**, **errmsg**)

Fault-tolerant alternative to `yRegisterHub()`.

YAPI.RegisterDeviceArrivalCallback(**arrivalCallback**)

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

YAPI.RegisterDeviceRemovalCallback(**removalCallback**)

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

YAPI.RegisterHub(**url**, **errmsg**)

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

YAPI.RegisterHubDiscoveryCallback(**hubDiscoveryCallback**)

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

YAPI.RegisterHubWebsocketCallback(**ws**, **errmsg**, **authpwd**)

Variant to `yRegisterHub()` used to initialize Yoctopuce API on an existing Websocket session, as happens for incoming WebSocket callbacks.

YAPI.RegisterLogFunction(**logfun**)

Registers a log callback function.

YAPI.SelectArchitecture(**arch**)

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

YAPI.SetCacheValidity(**cacheValidityMs**)

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

YAPI.SetDelegate(**object**)

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

YAPI.SetDeviceListValidity(**deviceListValidity**)

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

YAPI.SetHTTPCallbackCacheDir(**str_directory**)

Enables the HTTP callback cache.

YAPI.SetNetworkTimeout(**networkMsTimeout**)

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

YAPI.SetTimeout(**callback**, **ms_timeout**, **args**)

Invoke the specified callback function after a given timeout.

YAPI.SetUSBPacketAckMs(**pktAckDelay**)

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

YAPI.Sleep(**ms_duration**, **errmsg**)

Pauses the execution flow for a specified duration.

YAPI.TestHub(**url**, **mstimeout**, **errmsg**)

Test if the hub is reachable.

YAPI.TriggerHubDiscovery(**errmsg**)

Force a hub discovery, if a callback as been registered with `yRegisterHubDiscoveryCallback` it will be called for each net work hub that will respond to the discovery.

YAPI.UnregisterHub(**url**)

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

YAPI.UpdateDeviceList(**errmsg**)

Triggers a (re)detection of connected Yoctopuce modules.

YAPI.UpdateDeviceList_async(**callback**, **context**)

Triggers a (re)detection of connected Yoctopuce modules.

YAPI.CheckLogicalName() YAPI

YAPI.CheckLogicalName()

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

```
js      function yCheckLogicalName( name)
cpp     bool CheckLogicalName( string name)
m       +(BOOL) CheckLogicalName :(NSString *) name
pas     boolean yCheckLogicalName( name: string): boolean
vb      function CheckLogicalName( ByVal name As String) As Boolean
cs      static bool CheckLogicalName( string name)
java    boolean CheckLogicalName( String name)
uwp     bool CheckLogicalName( string name)
py      CheckLogicalName( name)
php     function CheckLogicalName( $name)
ts      async CheckLogicalName( name: string): Promise<boolean>
es      async CheckLogicalName( name)
```

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

Parameters :

name a string containing the name to check.

Returns :

`true` if the name is valid, `false` otherwise.

YAPI.ClearHTTPCallbackCacheDir() YAPI

YAPI.ClearHTTPCallbackCacheDir()

Disables the HTTP callback cache.

```
php function ClearHTTPCallbackCacheDir( $bool_removeFiles)
```

This method disables the HTTP callback cache, and can additionally cleanup the cache directory.

Parameters :

bool_removeFiles True to clear the content of the cache.

Returns :

nothing.

YAPI.DisableExceptions() YAPI
YAPI.DisableExceptions()

Disables the use of exceptions to report runtime errors.

js	function **yDisableExceptions**( )
cpp	void **DisableExceptions**( )
m	+(void) **DisableExceptions**
pas	**yDisableExceptions**( )
vb	procedure **DisableExceptions**( )
cs	static void **DisableExceptions**( )
uwp	void **DisableExceptions**( )
py	**DisableExceptions**( )
php	function **DisableExceptions**( )
ts	async **DisableExceptions**( ): Promise<void>
es	async **DisableExceptions**( )

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

YAPI.EnableExceptions() YAPI

YAPI.EnableExceptions()

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

js	function **yEnableExceptions**( )
cpp	void **EnableExceptions**( )
m	+(void) **EnableExceptions**
pas	**yEnableExceptions**( )
vb	procedure **EnableExceptions**( )
cs	static void **EnableExceptions**( )
uwp	void **EnableExceptions**( )
py	**EnableExceptions**( )
php	function **EnableExceptions**( )
ts	async **EnableExceptions**( ): Promise<void>
es	async **EnableExceptions**( )

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

YAPI.EnableUSBHost() YAPI
YAPI.EnableUSBHost()

This function is used only on Android.

```
java  void EnableUSBHost( Object osContext)
```

Before calling `yRegisterHub("usb")` you need to activate the USB host port of the system. This function takes as argument, an object of class android.content.Context (or any subclass). It is not necessary to call this function to reach modules through the network.

Parameters :

osContext an object of class android.content.Context (or any subclass).

YAPI.FreeAPI() YAPI
YAPI.FreeAPI()

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

js	function **yFreeAPI**( )
cpp	void **FreeAPI**( )
m	+(void) **FreeAPI**
pas	**yFreeAPI**( )
vb	procedure **FreeAPI**( )
cs	static void **FreeAPI**( )
java	void **FreeAPI**( )
uwp	void **FreeAPI**( )
py	**FreeAPI**( )
php	function **FreeAPI**( )
ts	async **FreeAPI**( ): Promise<void>
es	async **FreeAPI**( )
dnp	static void **FreeAPI**( )
cp	static void **FreeAPI**( )

From an operating system standpoint, it is generally not required to call this function since the OS will automatically free allocated resources once your program is completed. However there are two situations when you may really want to use that function: - Free all dynamically allocated memory blocks in order to track a memory leak. - Send commands to devices right before the end of the program. Since commands are sent in an asynchronous way the program could exit before all commands are effectively sent. You should not call any other library function after calling `yFreeAPI()`, or your program will crash.

YAPI.GetAPIVersion() YAPI

YAPI.GetAPIVersion()

Returns the version identifier for the Yoctopuce library in use.

js	function **yGetAPIVersion**( )
cpp	string **GetAPIVersion**( )
m	+(NSString*) **GetAPIVersion**
pas	string **yGetAPIVersion**( ): string
vb	function **GetAPIVersion**( ) As String
cs	static String **GetAPIVersion**( )
java	static String **GetAPIVersion**( )
uwp	static string **GetAPIVersion**( )
py	**GetAPIVersion**( )
php	function **GetAPIVersion**( )
ts	async **GetAPIVersion**( )
es	async **GetAPIVersion**( )
dnp	static string **GetAPIVersion**( )
cp	static string **GetAPIVersion**( )

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

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

Returns :
a character string describing the library version.

YAPI.GetCacheValidity() YAPI
YAPI.GetCacheValidity()

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

cpp	static u64 **GetCacheValidity**( )
m	+(u64) **GetCacheValidity**
pas	u64 **yGetCacheValidity**( ): u64
vb	function **GetCacheValidity**( ) As Long
cs	ulong **GetCacheValidity**( )
java	long **GetCacheValidity**( )
uwp	async Task<ulong> **GetCacheValidity**( )
py	**GetCacheValidity**( )
php	function **GetCacheValidity**( )
ts	async **GetCacheValidity**( ): Promise<number>
es	async **GetCacheValidity**( )

This method returns the cache validity of all attributes module functions. Note: This function must be called after `yInitAPI` .

Returns :
an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

YAPI.GetDeviceListValidity() YAPI
YAPI.GetDeviceListValidity()

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

cpp	static int **GetDeviceListValidity**( )
m	+(int) **GetDeviceListValidity**
pas	LongInt **yGetDeviceListValidity**( ): LongInt
vb	function **GetDeviceListValidity**( ) As Integer
cs	int **GetDeviceListValidity**( )
java	int **GetDeviceListValidity**( )
uwp	async Task<int> **GetDeviceListValidity**( )
py	**GetDeviceListValidity**( )
php	function **GetDeviceListValidity**( )
ts	async **GetDeviceListValidity**( ): Promise<number>
es	async **GetDeviceListValidity**( )

Note: you must call this function after `yInitAPI`.

Returns :
the number of seconds between each enumeration.

YAPI.GetDllArchitecture() YAPI
YAPI.GetDllArchitecture()

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

dnp static string **GetDllArchitecture**( )

On Windows, the architecture can be `"Win32"` or `"Win64"`. On ARM machines, the architecture is `"Armhf32"` or `"Aarch64"`. On other Linux machines, the architecture is `"Linux32"` or `"Linux64"`. On MacOS, the architecture is `"MacOs32"` or `"MacOs64"`.

Returns :
a character string describing the system architecture of the low-level communication library.

YAPI.GetDllPath() YAPI

YAPI.GetDllPath()

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

dnp static string **GetDllPath**( )

For architectures that require multiple DLLs, in particular when using a .NET assembly DLL, the returned string takes the form `"DotNetProxy=/...; yapi=/...;"`, where the first path corresponds to the .NET assembly DLL and the second path corresponds to the low-level communication library.

Returns :
a character string describing the DLL path.

YAPI.GetLog() YAPI

YAPI.GetLog()

Retrieves Yoctopuce low-level library diagnostic logs.

dnp static string **GetLog**(string **lastLogLine**)

cp static string **GetLog**(string **lastLogLine**)

This method allows to progessively retrieve API logs. The interface is line-based: it must called it within a loop until the returned value is an empty string. Make sure to exit the loop when an empty string is returned, as feeding an empty string into the `lastLogLine` parameter for the next call would restart enumerating logs from the oldest message available.

Parameters :

lastLogLine On first call, provide an empty string. On subsequent calls, provide the last log line returned by `GetLog()`.

Returns :

a string with the log line immediately following the one given in argument, if such line exist. Returns an empty string otherwise, when completed.

YAPI.GetNetworkTimeout() YAPI

YAPI.GetNetworkTimeout()

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

cpp	static int **GetNetworkTimeout**( )
m	+(int) **GetNetworkTimeout**
pas	LongInt **yGetNetworkTimeout**( ): LongInt
vb	function **GetNetworkTimeout**( ) As Integer
cs	int **GetNetworkTimeout**( )
java	int **GetNetworkTimeout**( )
uwp	async Task<int> **GetNetworkTimeout**( )
py	**GetNetworkTimeout**( )
php	function **GetNetworkTimeout**( )
ts	async **GetNetworkTimeout**( ): Promise<number>
es	async **GetNetworkTimeout**( )
dnp	static int **GetNetworkTimeout**( )
cp	static int **GetNetworkTimeout**( )

This delay impacts only the YoctoHubs and VirtualHub which are accessible through the network. By default, this delay is of 20000 milliseconds, but depending or you network you may want to change this delay, for example if your network infrastructure is based on a GSM connection.

Returns :
the network connection delay in milliseconds.

YAPI.GetTickCount() YAPI
YAPI.GetTickCount()

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

js	function **yGetTickCount**( )
cpp	u64 **GetTickCount**( )
m	+(u64) **GetTickCount**
pas	u64 **yGetTickCount**( ): u64
vb	function **GetTickCount**( ) As Long
cs	static ulong **GetTickCount**( )
java	static long **GetTickCount**( )
uwp	static ulong **GetTickCount**( )
py	**GetTickCount**( )
php	function **GetTickCount**( )
ts	**GetTickCount**( ): number
es	**GetTickCount**( )

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

Returns :
a long integer corresponding to the millisecond counter.

YAPI.HandleEvents() YAPI

YAPI.HandleEvents()

Maintains the device-to-library communication channel.

js	function **yHandleEvents**(**errmsg**)
cpp	YRETCODE **HandleEvents**(string **errmsg**)
m	+(YRETCODE) **HandleEvents** :(NSError**) **errmsg**
pas	integer **yHandleEvents**(var **errmsg**: string): integer
vb	function **HandleEvents**(ByRef **errmsg** As String) As YRETCODE
cs	static YRETCODE **HandleEvents**(ref string **errmsg**)
java	int **HandleEvents**( )
uwp	async Task<int> **HandleEvents**( )
py	**HandleEvents**(**errmsg**=None)
php	function **HandleEvents**(**&$errmsg**)
ts	async **HandleEvents**(**errmsg**: YErrorMsg | null): Promise<number>
es	async **HandleEvents**(**errmsg**)

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

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

Parameters :

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

Returns :

`YAPI.SUCCESS` when the call succeeds.

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

YAPI.InitAPI() YAPI

YAPI.InitAPI()

Initializes the Yoctopuce programming library explicitly.

```
js      function yInitAPI( mode, errmsg)
cpp     YRETCODE InitAPI( int mode, string errmsg)
m       +(YRETCODE) InitAPI :(int) mode :(NSError**) errmsg
pas     integer yInitAPI( mode: integer, var errmsg: string): integer
vb      function InitAPI( ByVal mode As Integer, ByRef errmsg As String) As Integer
cs      static int InitAPI( int mode, ref string errmsg)
java    int InitAPI( int mode)
uwp     async Task<int> InitAPI( int mode)
py      InitAPI( mode, errmsg=None)
php     function InitAPI( $mode, &$errmsg)
ts      async InitAPI( mode: number, errmsg: YErrorMsg): Promise<number>
es      async InitAPI( mode, errmsg)
```

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

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

Parameters :

mode an integer corresponding to the type of automatic device detection to use. Possible values are `YAPI.DETECT_NONE`, `YAPI.DETECT_USB`, `YAPI.DETECT_NET`, and `YAPI.DETECT_ALL`.

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

Returns :

`YAPI.SUCCESS` when the call succeeds.

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

YAPI.PreregisterHub() YAPI

YAPI.PreregisterHub()

Fault-tolerant alternative to `yRegisterHub()`.

```
js    function yPreregisterHub( url, errmsg)
cpp   YRETCODE PreregisterHub( string url, string errmsg)
m     +(YRETCODE) PreregisterHub :(NSString *) url :(NSError**) errmsg
pas   integer yPreregisterHub( url: string, var errmsg: string): integer
vb    function PreregisterHub( ByVal url As String,
                               ByRef errmsg As String) As Integer
cs    static int PreregisterHub( string url, ref string errmsg)
java  int PreregisterHub( String url)
uwp   async Task<int> PreregisterHub( string url)
py    PreregisterHub( url, errmsg=None)
php   function PreregisterHub( $url, &$errmsg)
ts    async PreregisterHub( url: string, errmsg: YErrorMsg): Promise<number>
es    async PreregisterHub( url, errmsg)
dnp   static string PreregisterHub( string url)
cp    static string PreregisterHub( string url)
```

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

Parameters :

url a string containing either **"usb"**,**"callback"** or the root URL of the hub to monitor

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

Returns :

`YAPI.SUCCESS` when the call succeeds.

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

YAPI.RegisterDeviceArrivalCallback() YAPI
YAPI.RegisterDeviceArrivalCallback()

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

js	function **yRegisterDeviceArrivalCallback**(**arrivalCallback**)
cpp	void **RegisterDeviceArrivalCallback**(yDeviceUpdateCallback **arrivalCallback**)
m	+(void) **RegisterDeviceArrivalCallback** :(yDeviceUpdateCallback) **arrivalCallback**
pas	**yRegisterDeviceArrivalCallback**(**arrivalCallback**: yDeviceUpdateFunc)
vb	procedure **RegisterDeviceArrivalCallback**(ByVal **arrivalCallback** As yDeviceUpdateFunc)
cs	static void **RegisterDeviceArrivalCallback**(yDeviceUpdateFunc **arrivalCallback**)
java	void **RegisterDeviceArrivalCallback**(DeviceArrivalCallback **arrivalCallback**)
uwp	void **RegisterDeviceArrivalCallback**(DeviceUpdateHandler **arrivalCallback**)
py	**RegisterDeviceArrivalCallback**(**arrivalCallback**)
php	function **RegisterDeviceArrivalCallback**(**$arrivalCallback**)
ts	async **RegisterDeviceArrivalCallback**(**arrivalCallback**: YDeviceUpdateCallback| null): Promise<void>
es	async **RegisterDeviceArrivalCallback**(**arrivalCallback**)

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

Parameters :

arrivalCallback a procedure taking a `YModule` parameter, or `null`

YAPI.RegisterDeviceRemovalCallback() YAPI
YAPI.RegisterDeviceRemovalCallback()

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

js	function **yRegisterDeviceRemovalCallback**(**removalCallback**)
cpp	void **RegisterDeviceRemovalCallback**(yDeviceUpdateCallback **removalCallback**)
m	+(void) **RegisterDeviceRemovalCallback** :(yDeviceUpdateCallback) **removalCallback**
pas	**yRegisterDeviceRemovalCallback**(**removalCallback**: yDeviceUpdateFunc)
vb	procedure **RegisterDeviceRemovalCallback**(ByVal **removalCallback** As yDeviceUpdateFunc)
cs	static void **RegisterDeviceRemovalCallback**(yDeviceUpdateFunc **removalCallback**)
java	void **RegisterDeviceRemovalCallback**(DeviceRemovalCallback **removalCallback**)
uwp	void **RegisterDeviceRemovalCallback**(DeviceUpdateHandler **removalCallback**)
py	**RegisterDeviceRemovalCallback**(**removalCallback**)
php	function **RegisterDeviceRemovalCallback**(**$removalCallback**)
ts	async **RegisterDeviceRemovalCallback**(**removalCallback**: YDeviceUpdateCallback| null): Promise<void>
es	async **RegisterDeviceRemovalCallback**(**removalCallback**)

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

Parameters :

removalCallback a procedure taking a `YModule` parameter, or `null`

YAPI.RegisterHub() YAPI
YAPI.RegisterHub()

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

```
js      function yRegisterHub( url, errmsg)
cpp     YRETCODE RegisterHub( string url, string errmsg)
m       +(YRETCODE) RegisterHub :(NSString *) url :(NSError**) errmsg
pas     integer yRegisterHub( url: string, var errmsg: string): integer
vb      function RegisterHub( ByVal url As String,
                              ByRef errmsg As String) As Integer
cs      static int RegisterHub( string url, ref string errmsg)
java    int RegisterHub( String url)
uwp     async Task<int> RegisterHub( string url)
py      RegisterHub( url, errmsg=None)
php     function RegisterHub( $url, &$errmsg)
ts      async RegisterHub( url: string, errmsg: YErrorMsg): Promise<number>
es      async RegisterHub( url, errmsg)
dnp     static string RegisterHub( string url)
cp      static string RegisterHub( string url)
```

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

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

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

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

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

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

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

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

Parameters :

url a string containing either **"usb"**,**"callback"** or the root URL of the hub to monitor

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

Returns :
YAPI.SUCCESS when the call succeeds.

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

YAPI.RegisterHubDiscoveryCallback() YAPI
YAPI.RegisterHubDiscoveryCallback()

Register a callback function, to be called each time an Network Hub send an SSDP message.

cpp	void **RegisterHubDiscoveryCallback**(YHubDiscoveryCallback **hubDiscoveryCallback**)
m	+(void) **RegisterHubDiscoveryCallback** : (YHubDiscoveryCallback) **hubDiscoveryCallback**
pas	**yRegisterHubDiscoveryCallback**(**hubDiscoveryCallback**: YHubDiscoveryCallback)
vb	procedure **RegisterHubDiscoveryCallback**(ByVal **hubDiscoveryCallback** As YHubDiscoveryCallback)
cs	static void **RegisterHubDiscoveryCallback**(YHubDiscoveryCallback **hubDiscoveryCallback**)
java	void **RegisterHubDiscoveryCallback**(HubDiscoveryCallback **hubDiscoveryCallback**)
uwp	async Task **RegisterHubDiscoveryCallback**(HubDiscoveryHandler **hubDiscoveryCallback**)
py	**RegisterHubDiscoveryCallback**(**hubDiscoveryCallback**)
ts	async **RegisterHubDiscoveryCallback**(**hubDiscoveryCallback**: YHubDiscoveryCallback): Promise<number>
es	async **RegisterHubDiscoveryCallback**(**hubDiscoveryCallback**)

The callback has two string parameter, the first one contain the serial number of the hub and the second contain the URL of the network hub (this URL can be passed to RegisterHub). This callback will be invoked while yUpdateDeviceList is running. You will have to call this function on a regular basis.

Parameters :

hubDiscoveryCallback a procedure taking two string parameter, the serial

YAPI.RegisterHubWebsocketCallback() YAPI
YAPI.RegisterHubWebsocketCallback()

Variant to `yRegisterHub()` used to initialize Yoctopuce API on an existing Websocket session, as happens for incoming WebSocket callbacks.

Parameters :

ws	node WebSocket object for the incoming WebSocket callback connection
errmsg	a string passed by reference to receive any error message.
authpwd	the optional authentication password, required only authentication is configured on the calling hub.

Returns :

`YAPI.SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

YAPI.RegisterLogFunction() YAPI

YAPI.RegisterLogFunction()

Registers a log callback function.

cpp	void **RegisterLogFunction**(yLogFunction **logfun**)
m	+(void) **RegisterLogFunction** :(yLogCallback) **logfun**
pas	**yRegisterLogFunction**(**logfun**: yLogFunc)
vb	procedure **RegisterLogFunction**(ByVal **logfun** As yLogFunc)
cs	static void **RegisterLogFunction**(yLogFunc **logfun**)
java	void **RegisterLogFunction**(LogCallback **logfun**)
uwp	void **RegisterLogFunction**(LogHandler **logfun**)
py	**RegisterLogFunction**(**logfun**)
ts	async **RegisterLogFunction**(**logfun**: YLogCallback): Promise<number>
es	async **RegisterLogFunction**(**logfun**)

This callback will be called each time the API have something to say. Quite useful to debug the API.

Parameters :

logfun a procedure taking a string parameter, or `null`

YAPI.SelectArchitecture() YAPI

YAPI.SelectArchitecture()

Select the architecture or the library to be loaded to access to USB.

py **SelectArchitecture**(**arch**)

By default, the Python library automatically detects the appropriate library to use. However, for Linux ARM, it not possible to reliably distinguish between a Hard Float (armhf) and a Soft Float (armel) install. For in this case, it is therefore recommended to manually select the proper architecture by calling `SelectArchitecture()` before any other call to the library.

Parameters :

arch A string containing the architecture to use. Possibles value are: `"armhf"`,`"armel"`, `"aarch64"`,`"i386"`,`"x86_64"`, `"32bit"`, `"64bit"`

Returns :

nothing.

On failure, throws an exception.

YAPI.SetCacheValidity() YAPI
YAPI.SetCacheValidity()

Change the validity period of the data loaded by the library.

cpp	static void **SetCacheValidity**(u64 **cacheValidityMs**)
m	+(void) **SetCacheValidity** : (u64) **cacheValidityMs**
pas	**ySetCacheValidity**(**cacheValidityMs**: u64)
vb	procedure **SetCacheValidity**(ByVal **cacheValidityMs** As Long)
cs	void **SetCacheValidity**(ulong **cacheValidityMs**)
java	void **SetCacheValidity**(long **cacheValidityMs**)
uwp	async Task **SetCacheValidity**(ulong **cacheValidityMs**)
py	**SetCacheValidity**(**cacheValidityMs**)
php	function **SetCacheValidity**(**$cacheValidityMs**)
ts	async **SetCacheValidity**(**cacheValidityMs**: number): Promise<void>
es	async **SetCacheValidity**(**cacheValidityMs**)

By default, when accessing a module, all the attributes of the module functions are automatically kept in cache for the standard duration (5 ms). This method can be used to change this standard duration, for example in order to reduce network or USB traffic. This parameter does not affect value change callbacks Note: This function must be called after `yInitAPI`.

Parameters :

cacheValidityMs an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds.

YAPI.SetDelegate() YAPI
YAPI.SetDelegate()

(Objective-C only) Register an object that must follow the protocol YDeviceHotPlug.

m +(void) **SetDelegate** :(id) **object**

The methods yDeviceArrival and yDeviceRemoval will be invoked while yUpdateDeviceList is running. You will have to call this function on a regular basis.

Parameters :

object an object that must follow the protocol YAPIDelegate, or nil

YAPI.SetDeviceListValidity() YAPI
YAPI.SetDeviceListValidity()

Modifies the delay between each forced enumeration of the used YoctoHubs.

cpp	static void **SetDeviceListValidity**(int **deviceListValidity**)
m	+(void) **SetDeviceListValidity** : (int) **deviceListValidity**
pas	**ySetDeviceListValidity**(**deviceListValidity**: LongInt)
vb	procedure **SetDeviceListValidity**(ByVal **deviceListValidity** As Integer)
cs	void **SetDeviceListValidity**(int **deviceListValidity**)
java	void **SetDeviceListValidity**(int **deviceListValidity**)
uwp	async Task **SetDeviceListValidity**(int **deviceListValidity**)
py	**SetDeviceListValidity**(**deviceListValidity**)
php	function **SetDeviceListValidity**(**$deviceListValidity**)
ts	async **SetDeviceListValidity**(**deviceListValidity**: number): Promise<void>
es	async **SetDeviceListValidity**(**deviceListValidity**)

By default, the library performs a full enumeration every 10 seconds. To reduce network traffic, you can increase this delay. It's particularly useful when a YoctoHub is connected to the GSM network where traffic is billed. This parameter doesn't impact modules connected by USB, nor the working of module arrival/removal callbacks. Note: you must call this function after `yInitAPI`.

Parameters :

deviceListValidity nubmer of seconds between each enumeration.

YAPI.SetHTTPCallbackCacheDir() YAPI
YAPI.SetHTTPCallbackCacheDir()

Enables the HTTP callback cache.

```
php function SetHTTPCallbackCacheDir( $str_directory)
```

When enabled, this cache reduces the quantity of data sent to the PHP script by 50% to 70%. To enable this cache, the method `ySetHTTPCallbackCacheDir()` must be called before any call to `yRegisterHub()`. This method takes in parameter the path of the directory used for saving data between each callback. This folder must exist and the PHP script needs to have write access to it. It is recommended to use a folder that is not published on the Web server since the library will save some data of Yoctopuce devices into this folder.

Note: This feature is supported by YoctoHub and VirtualHub since version 27750.

Parameters :

str_directory the path of the folder that will be used as cache.

Returns :

nothing.

On failure, throws an exception.

YAPI.SetNetworkTimeout() YAPI

YAPI.SetNetworkTimeout()

Modifies the network connection delay for `yRegisterHub()` and `yUpdateDeviceList()`.

cpp	static void **SetNetworkTimeout**(int **networkMsTimeout**)
m	+(void) **SetNetworkTimeout** : (int) **networkMsTimeout**
pas	**ySetNetworkTimeout**(**networkMsTimeout**: LongInt)
vb	procedure **SetNetworkTimeout**(ByVal **networkMsTimeout** As Integer)
cs	void **SetNetworkTimeout**(int **networkMsTimeout**)
java	void **SetNetworkTimeout**(int **networkMsTimeout**)
uwp	async Task **SetNetworkTimeout**(int **networkMsTimeout**)
py	**SetNetworkTimeout**(**networkMsTimeout**)
php	function **SetNetworkTimeout**(**$networkMsTimeout**)
ts	async **SetNetworkTimeout**(**networkMsTimeout**: number): Promise<void>
es	async **SetNetworkTimeout**(**networkMsTimeout**)
dnp	static void **SetNetworkTimeout**(int **networkMsTimeout**)
cp	static void **SetNetworkTimeout**(int **networkMsTimeout**)

This delay impacts only the YoctoHubs and VirtualHub which are accessible through the network. By default, this delay is of 20000 milliseconds, but depending or you network you may want to change this delay, gor example if your network infrastructure is based on a GSM connection.

Parameters :

networkMsTimeout the network connection delay in milliseconds.

YAPI.SetTimeout() YAPI

YAPI.SetTimeout()

Invoke the specified callback function after a given timeout.

```
js  function ySetTimeout( callback, ms_timeout, args)
ts  SetTimeout( callback: Function, ms_timeout: number): number
es  SetTimeout( callback, ms_timeout, args)
```

This function behaves more or less like Javascript `setTimeout`, but during the waiting time, it will call `yHandleEvents` and `yUpdateDeviceList` periodically, in order to keep the API up-to-date with current devices.

Parameters :

callback	the function to call after the timeout occurs. On Microsoft Internet Explorer, the callback must be provided as a string to be evaluated.
ms_timeout	an integer corresponding to the duration of the timeout, in milliseconds.
args	additional arguments to be passed to the callback function can be provided, if needed (not supported on Microsoft Internet Explorer).

Returns :

`YAPI.SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

YAPI.SetUSBPacketAckMs() YAPI

YAPI.SetUSBPacketAckMs()

Enables the acknowledge of every USB packet received by the Yoctopuce library.

```
java  void SetUSBPacketAckMs( int pktAckDelay)
```

This function allows the library to run on Android phones that tend to loose USB packets. By default, this feature is disabled because it doubles the number of packets sent and slows down the API considerably. Therefore, the acknowledge of incoming USB packets should only be enabled on phones or tablets that loose USB packets. A delay of 50 milliseconds is generally enough. In case of doubt, contact Yoctopuce support. To disable USB packets acknowledge, call this function with the value 0. Note: this feature is only available on Android.

Parameters :

pktAckDelay then number of milliseconds before the module

YAPI.Sleep() YAPI

YAPI.Sleep()

Pauses the execution flow for a specified duration.

```
js    function ySleep( ms_duration, errmsg)
cpp   YRETCODE Sleep( unsigned ms_duration, string errmsg)
m     +(YRETCODE) Sleep :(unsigned) ms_duration :(NSError **) errmsg
pas   integer ySleep( ms_duration: integer, var errmsg: string): integer
vb    function Sleep( ByVal ms_duration As Integer,
                      ByRef errmsg As String) As Integer
cs    static int Sleep( int ms_duration, ref string errmsg)
java  int Sleep( long ms_duration)
uwp   async Task<int> Sleep( ulong ms_duration)
py    Sleep( ms_duration, errmsg=None)
php   function Sleep( $ms_duration, &$errmsg)
ts    async Sleep( ms_duration: number, errmsg: YErrorMsg | null): Promise<number>
es    async Sleep( ms_duration, errmsg)
```

This function implements a passive waiting loop, meaning that it does not consume CPU cycles significantly. The processor is left available for other threads and processes. During the pause, the library nevertheless reads from time to time information from the Yoctopuce modules by calling `yHandleEvents()`, in order to stay up-to-date.

This function may signal an error in case there is a communication problem while contacting a module.

Parameters :

ms_duration an integer corresponding to the duration of the pause, in milliseconds.

errmsg a string passed by reference to receive any error message.

Returns :

`YAPI.SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

YAPI.TestHub() YAPI

YAPI.TestHub()

Test if the hub is reachable.

```
cpp     YRETCODE TestHub( string url, int mstimeout, string errmsg)
m       +(YRETCODE) TestHub : (NSString*) url
                            : (int) mstimeout
                            : (NSError**) errmsg
pas     integer yTestHub( url: string,
                          mstimeout: integer,
                          var errmsg: string): integer
vb      function TestHub( ByVal url As String,
                          ByVal mstimeout As Integer,
                          ByRef errmsg As String) As Integer
cs      static int TestHub( string url, int mstimeout, ref string errmsg)
java    int TestHub( String url, int mstimeout)
uwp     async Task<int> TestHub( string url, uint mstimeout)
py      TestHub( url, mstimeout, errmsg=None)
php     function TestHub( $url, $mstimeout, &$errmsg)
ts      async TestHub( url: string, mstimeout: number, errmsg: YErrorMsg): Promise<number>
es      async TestHub( url, mstimeout, errmsg)
dnp     static string TestHub( string url, int mstimeout)
cp      static string TestHub( string url, int mstimeout)
```

This method do not register the hub, it only test if the hub is usable. The url parameter follow the same convention as the `yRegisterHub` method. This method is useful to verify the authentication parameters for a hub. It is possible to force this method to return after mstimeout milliseconds.

Parameters :

url	a string containing either **"usb"**,**"callback"** or the root URL of the hub to monitor
mstimeout	the number of millisecond available to test the connection.
errmsg	a string passed by reference to receive any error message.

Returns :

`YAPI.SUCCESS` when the call succeeds.

On failure returns a negative error code.

YAPI.TriggerHubDiscovery() YAPI
YAPI.TriggerHubDiscovery()

Force a hub discovery, if a callback as been registered with `yRegisterHubDiscoveryCallback` it will be called for each net work hub that will respond to the discovery.

cpp	YRETCODE **TriggerHubDiscovery**(string **errmsg**)
m	+(YRETCODE) **TriggerHubDiscovery** : (NSError**) **errmsg**
pas	integer **yTriggerHubDiscovery**(var **errmsg**: string): integer
vb	function **TriggerHubDiscovery**(ByRef **errmsg** As String) As Integer
cs	static int **TriggerHubDiscovery**(ref string **errmsg**)
java	int **TriggerHubDiscovery**( )
uwp	Task<int> **TriggerHubDiscovery**( )
py	**TriggerHubDiscovery**(**errmsg**=None)
ts	async **TriggerHubDiscovery**(**errmsg**: YErrorMsg | null): Promise<number>
es	async **TriggerHubDiscovery**(**errmsg**)

Parameters :

errmsg a string passed by reference to receive any error message.

Returns :

`YAPI.SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

YAPI.UnregisterHub() YAPI

YAPI.UnregisterHub()

Setup the Yoctopuce library to no more use modules connected on a previously registered machine with RegisterHub.

js	function **yUnregisterHub**(**url**)
cpp	void **UnregisterHub**(string **url**)
m	+(void) **UnregisterHub** :(NSString *) **url**
pas	**yUnregisterHub**(**url**: string)
vb	procedure **UnregisterHub**(ByVal **url** As String)
cs	static void **UnregisterHub**(string **url**)
java	void **UnregisterHub**(String **url**)
uwp	async Task **UnregisterHub**(string **url**)
py	**UnregisterHub**(**url**)
php	function **UnregisterHub**(**$url**)
ts	async **UnregisterHub**(**url**: string): Promise<void>
es	async **UnregisterHub**(**url**)

Parameters :

url a string containing either **"usb"** or the

YAPI.UpdateDeviceList() YAPI

YAPI.UpdateDeviceList()

Triggers a (re)detection of connected Yoctopuce modules.

js	function **yUpdateDeviceList**(**errmsg**)
cpp	YRETCODE **UpdateDeviceList**(string **errmsg**)
m	+(YRETCODE) **UpdateDeviceList** :(NSError**) **errmsg**
pas	integer **yUpdateDeviceList**(var **errmsg**: string): integer
vb	function **UpdateDeviceList**(ByRef **errmsg** As String) As YRETCODE
cs	static YRETCODE **UpdateDeviceList**(ref string **errmsg**)
java	int **UpdateDeviceList**( )
uwp	async Task<int> **UpdateDeviceList**( )
py	**UpdateDeviceList**(**errmsg**=None)
php	function **UpdateDeviceList**(**&$errmsg**)
ts	async **UpdateDeviceList**(**errmsg**: YErrorMsg | null): Promise<number>
es	async **UpdateDeviceList**(**errmsg**)

The library searches the machines or USB ports previously registered using `yRegisterHub()`, and invokes any user-defined callback function in case a change in the list of connected devices is detected.

This function can be called as frequently as desired to refresh the device list and to make the application aware of hot-plug events. However, since device detection is quite a heavy process, UpdateDeviceList shouldn't be called more than once every two seconds.

Parameters :

errmsg a string passed by reference to receive any error message.

Returns :

`YAPI.SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

YAPI.UpdateDeviceList_async() YAPI

YAPI.UpdateDeviceList_async()

Triggers a (re)detection of connected Yoctopuce modules.

```
js  function yUpdateDeviceList_async( callback, context)
```

The library searches the machines or USB ports previously registered using `yRegisterHub()`, and invokes any user-defined callback function in case a change in the list of connected devices is detected.

This function can be called as frequently as desired to refresh the device list and to make the application aware of hot-plug events.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking Firefox JavaScript VM that does not implement context switching during blocking I/O calls.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the result code (`YAPI.SUCCESS` if the operation completes successfully) and the error message.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

24.2. Class YModule

Global parameters control interface for all Yoctopuce devices

The `YModule` class can be used with all Yoctopuce USB devices. It can be used to control the module global parameters, and to enumerate the functions provided by each module.

In order to use the functions described here, you should include:

`js`	<script type='text/javascript' src='yocto_api.js'></script>
`cpp`	#include "yocto_api.h"
`m`	#import "yocto_api.h"
`pas`	uses yocto_api;
`vb`	yocto_api.vb
`cs`	yocto_api.cs
`java`	import com.yoctopuce.YoctoAPI.YModule;
`uwp`	import com.yoctopuce.YoctoAPI.YModule;
`py`	from yocto_api import *
`php`	require_once('yocto_api.php');
`ts`	in HTML: import { YAPI, YErrorMsg, YModule, YSensor } from '../../dist/esm/yocto_api_browser.js'; in Node.js: import { YAPI, YErrorMsg, YModule, YSensor } from 'yoctolib-cjs/yocto_api_nodejs.js';
`es`	in HTML: <script src="../../lib/yocto_api.js"></script> in node.js: require('yoctolib-es2017/yocto_api.js');
`dnp`	import YoctoProxyAPI.YModuleProxy
`cp`	#include "yocto_module_proxy.h"
`vi`	YModule.vi
`ml`	import YoctoProxyAPI.YModuleProxy"

Global functions

YModule.FindModule(**func**)

Allows you to find a module from its serial number or from its logical name.

YModule.FindModuleInContext(**yctx**, **func**)

Retrieves a module for a given identifier in a YAPI context.

YModule.FirstModule()

Starts the enumeration of modules currently accessible.

`YModule` properties

module→Beacon *[writable]*

State of the localization beacon.

module→FirmwareRelease *[read-only]*

Version of the firmware embedded in the module.

module→FunctionId *[read-only]*

Retrieves the hardware identifier of the *n*th function on the module.

module→HardwareId *[read-only]*

Unique hardware identifier of the module.

module→IsOnline *[read-only]*

Checks if the module is currently reachable.

module→LogicalName *[writable]*

Logical name of the module.

module→Luminosity *[writable]*

Luminosity of the module informative LEDs (from 0 to 100).

module→ProductId *[read-only]*

USB device identifier of the module.

module→ProductName *[read-only]*

Commercial name of the module, as set by the factory.

module→ProductRelease *[read-only]*

Release number of the module hardware, preprogrammed at the factory.

module→SerialNumber *[read-only]*

Serial number of the module, as set by the factory.

`YModule` methods

module→checkFirmware(path, onlynew)

Tests whether the byn file is valid for this module.

module→clearCache()

Invalidates the cache.

module→describe()

Returns a descriptive text that identifies the module.

module→download(pathname)

Downloads the specified built-in file and returns a binary buffer with its content.

module→functionBaseType(functionIndex)

Retrieves the base type of the *n*th function on the module.

module→functionCount()

Returns the number of functions (beside the "module" interface) available on the module.

module→functionId(functionIndex)

Retrieves the hardware identifier of the *n*th function on the module.

module→functionName(functionIndex)

Retrieves the logical name of the *n*th function on the module.

module→functionType(functionIndex)

Retrieves the type of the *n*th function on the module.

module→functionValue(functionIndex)

Retrieves the advertised value of the *n*th function on the module.

module→get_allSettings()

Returns all the settings and uploaded files of the module.

module→get_beacon()

Returns the state of the localization beacon.

module→get_errorMessage()

Returns the error message of the latest error with this module object.

module→get_errorType()

Returns the numerical error code of the latest error with this module object.

module→get_firmwareRelease()

Returns the version of the firmware embedded in the module.

module→get_functionIds(funType)

Retrieve all hardware identifier that match the type passed in argument.

module→get_hardwareId()

Returns the unique hardware identifier of the module.

module→get_icon2d()

Returns the icon of the module.

module→get_lastLogs()
Returns a string with last logs of the module.

module→get_logicalName()
Returns the logical name of the module.

module→get_luminosity()
Returns the luminosity of the module informative LEDs (from 0 to 100).

module→get_parentHub()
Returns the serial number of the YoctoHub on which this module is connected.

module→get_persistentSettings()
Returns the current state of persistent module settings.

module→get_productId()
Returns the USB device identifier of the module.

module→get_productName()
Returns the commercial name of the module, as set by the factory.

module→get_productRelease()
Returns the release number of the module hardware, preprogrammed at the factory.

module→get_rebootCountdown()
Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

module→get_serialNumber()
Returns the serial number of the module, as set by the factory.

module→get_subDevices()
Returns a list of all the modules that are plugged into the current module.

module→get_upTime()
Returns the number of milliseconds spent since the module was powered on.

module→get_url()
Returns the URL used to access the module.

module→get_usbCurrent()
Returns the current consumed by the module on the USB bus, in milli-amps.

module→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

module→get_userVar()
Returns the value previously stored in this attribute.

module→hasFunction(**funcId**)
Tests if the device includes a specific function.

module→isOnline()
Checks if the module is currently reachable, without raising any error.

module→isOnline_async(**callback**, **context**)
Checks if the module is currently reachable, without raising any error.

module→load(**msValidity**)
Preloads the module cache with a specified validity duration.

module→load_async(**msValidity**, **callback**, **context**)
Preloads the module cache with a specified validity duration (asynchronous version).

module→log(**text**)
Adds a text message to the device logs.

module→nextModule()

Continues the module enumeration started using `yFirstModule()`.

module→reboot(**secBeforeReboot**)

Schedules a simple module reboot after the given number of seconds.

module→registerBeaconCallback(**callback**)

Register a callback function, to be called when the localization beacon of the module has been changed.

module→registerConfigChangeCallback(**callback**)

Register a callback function, to be called when a persistent settings in a device configuration has been changed (e.g.

module→registerLogCallback(**callback**)

Registers a device log callback function.

module→revertFromFlash()

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

module→saveToFlash()

Saves current settings in the nonvolatile memory of the module.

module→set_allSettings(**settings**)

Restores all the settings of the device.

module→set_allSettingsAndFiles(**settings**)

Restores all the settings and uploaded files to the module.

module→set_beacon(**newval**)

Turns on or off the module localization beacon.

module→set_logicalName(**newval**)

Changes the logical name of the module.

module→set_luminosity(**newval**)

Changes the luminosity of the module informative leds.

module→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

module→set_userVar(**newval**)

Stores a 32 bit value in the device RAM.

module→triggerConfigChangeCallback()

Triggers a configuration change callback, to check if they are supported or not.

module→triggerFirmwareUpdate(**secBeforeReboot**)

Schedules a module reboot into special firmware update mode.

module→updateFirmware(**path**)

Prepares a firmware update of the module.

module→updateFirmwareEx(**path**, **force**)

Prepares a firmware update of the module.

module→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YModule.FindModule() YModule
YModule.FindModule()

Allows you to find a module from its serial number or from its logical name.

js	function **yFindModule**(**func**)
cpp	YModule* **FindModule**(string **func**)
m	+(YModule*) **FindModule** : (NSString*) **func**
pas	TYModule **yFindModule**(**func**: string): TYModule
vb	function **FindModule**(ByVal **func** As String) As YModule
cs	static YModule **FindModule**(string **func**)
java	static YModule **FindModule**(String **func**)
uwp	static YModule **FindModule**(string **func**)
py	**FindModule**(**func**)
php	function **FindModule**(**$func**)
ts	static **FindModule**(**func**: string): YModule
es	static **FindModule**(**func**)
dnp	static YModuleProxy **FindModule**(string **func**)
cp	static YModuleProxy * **FindModule**(string **func**)

This function does not require that the module is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YModule.isOnline()` to test if the module is indeed online at a given time. In case of ambiguity when looking for a module by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

If a call to this object's is_online() method returns FALSE although you are certain that the device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string containing either the serial number or the logical name of the desired module

Returns :

a `YModule` object allowing you to drive the module or get additional information on the module.

YModule.FindModuleInContext() YModule YModule.FindModuleInContext()

Retrieves a module for a given identifier in a YAPI context.

```
java  static YModule FindModuleInContext( YAPIContext yctx, String func)
uwp   static YModule FindModuleInContext( YAPIContext yctx, string func)
ts    static FindModuleInContext( yctx: YAPIContext, func: string): YModule
es    static FindModuleInContext( yctx, func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the module is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YModule.isOnline()` to test if the module is indeed online at a given time. In case of ambiguity when looking for a module by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the module, for instance `MyDevice.module`.

Returns :

a `YModule` object allowing you to drive the module.

YModule.FirstModule() YModule
YModule.FirstModule()

Starts the enumeration of modules currently accessible.

js	function **yFirstModule**( )
cpp	YModule * **FirstModule**( )
m	+(YModule*) **FirstModule**
pas	TYModule **yFirstModule**( ): TYModule
vb	function **FirstModule**( ) As YModule
cs	static YModule **FirstModule**( )
java	static YModule **FirstModule**( )
uwp	static YModule **FirstModule**( )
py	**FirstModule**( )
php	function **FirstModule**( )
ts	static **FirstModule**( ): YModule | null
es	static **FirstModule**( )

Use the method `YModule.nextModule()` to iterate on the next modules.

Returns :

a pointer to a `YModule` object, corresponding to the first module currently online, or a `null` pointer if there are none.

module→Beacon YModule

State of the localization beacon.

dnp int **Beacon**

Writable. Turns on or off the module localization beacon.

module→FirmwareRelease YModule

Version of the firmware embedded in the module.

dnp string **FirmwareRelease**

module→FunctionId — YModule

Retrieves the hardware identifier of the *n*th function on the module.

dnp string **FunctionId**

@param functionIndex : the index of the function for which the information is desired, starting at 0 for the first function.

module→HardwareId — YModule

Unique hardware identifier of the module.

dnp string **HardwareId**

The unique hardware identifier is made of the device serial number followed by string ".module".

module→IsOnline — YModule

Checks if the module is currently reachable.

dnp bool **IsOnline**

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

module→LogicalName — YModule

Logical name of the module.

dnp string **LogicalName**

Writable. You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

module→Luminosity YModule

Luminosity of the module informative LEDs (from 0 to 100).

dnp int **Luminosity**

Writable. Changes the luminosity of the module informative leds. The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

module→ProductId — YModule

USB device identifier of the module.

dnp int **ProductId**

module→ProductName — YModule

Commercial name of the module, as set by the factory.

dnp string **ProductName**

module→ProductRelease YModule

Release number of the module hardware, preprogrammed at the factory.

dnp int **ProductRelease**

The original hardware release returns value 1, revision B returns value 2, etc.

module→SerialNumber — YModule

Serial number of the module, as set by the factory.

dnp string **SerialNumber**

module→checkFirmware() YModule

Tests whether the byn file is valid for this module.

```
js      function checkFirmware( path, onlynew)
cpp     string checkFirmware( string path, bool onlynew)
m       -(NSString*) checkFirmware : (NSString*) path
                                   : (bool) onlynew
pas     string checkFirmware( path: string, onlynew: boolean): string
vb      function checkFirmware( ByVal path As String, ByVal onlynew As Boolean) As String
cs      string checkFirmware( string path, bool onlynew)
java    String checkFirmware( String path, boolean onlynew)
uwp     async Task<string> checkFirmware( string path, bool onlynew)
py      checkFirmware( path, onlynew)
php     function checkFirmware( $path, $onlynew)
ts      async checkFirmware( path: string, onlynew: boolean): Promise<string>
es      async checkFirmware( path, onlynew)
dnp     string checkFirmware( string path, bool onlynew)
cp      string checkFirmware( string path, bool onlynew)
cmd     YModule target checkFirmware path onlynew
```

This method is useful to test if the module needs to be updated. It is possible to pass a directory as argument instead of a file. In this case, this method returns the path of the most recent appropriate `.byn` file. If the parameter `onlynew` is true, the function discards firmwares that are older or equal to the installed firmware.

Parameters :

path the path of a byn file or a directory that contains byn files

onlynew returns only files that are strictly newer

Returns :

the path of the byn file to use or a empty string if no byn files matches the requirement

On failure, throws an exception or returns a string that start with "error:".

module→clearCache() YModule

Invalidates the cache.

js	function **clearCache**( )
cpp	void **clearCache**( )
m	-(void) **clearCache**
pas	**clearCache**( )
vb	procedure **clearCache**( )
cs	void **clearCache**( )
java	void **clearCache**( )
py	**clearCache**( )
php	function **clearCache**( )
ts	async **clearCache**( ): Promise<void>
es	async **clearCache**( )

Invalidates the cache of the module attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

module→describe() YModule

Returns a descriptive text that identifies the module.

```
js      function describe( )
cpp     string describe( )
m       -(NSString*) describe
pas     string describe( ): string
vb      function describe( ) As String
cs      string describe( )
java    String describe( )
py      describe( )
php     function describe( )
ts      async describe( ): Promise<string>
es      async describe( )
```

The text may include either the logical name or the serial number of the module.

Returns :
a string that describes the module

module→download() YModule

Downloads the specified built-in file and returns a binary buffer with its content.

```
js      function download( pathname)
cpp     string download( string pathname)
m       -(NSMutableData*) download : (NSString*) pathname
pas     TByteArray download( pathname: string): TByteArray
vb      function download( ByVal pathname As String) As Byte
cs      byte[] download( string pathname)
java    byte[] download( String pathname)
uwp     async Task<byte[]> download( string pathname)
py      download( pathname)
php     function download( $pathname)
ts      async download( pathname: string): Promise<Uint8Array>
es      async download( pathname)
dnp     byte[] download( string pathname)
cp      string download( string pathname)
cmd     YModule target download pathname
```

Parameters :

pathname name of the new file to load

Returns :

a binary buffer with the file content

On failure, throws an exception or returns `YAPI_INVALID_STRING`.

module→functionBaseType() YModule

Retrieves the base type of the *n*th function on the module.

```
js      function functionBaseType( functionIndex)
cpp     string functionBaseType( int functionIndex)
pas     string functionBaseType( functionIndex: integer): string
vb      function functionBaseType( ByVal functionIndex As Integer) As String
cs      string functionBaseType( int functionIndex)
java    String functionBaseType( int functionIndex)
py      functionBaseType( functionIndex)
php     function functionBaseType( $functionIndex)
ts      async functionBaseType( functionIndex: number): Promise<string>
es      async functionBaseType( functionIndex)
```

For instance, the base type of all measuring functions is "Sensor".

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a string corresponding to the base type of the function

On failure, throws an exception or returns an empty string.

module→functionCount() YModule

Returns the number of functions (beside the "module" interface) available on the module.

```
js      function functionCount( )
cpp     int functionCount( )
m       -(int) functionCount
pas     integer functionCount( ): integer
vb      function functionCount( ) As Integer
cs      int functionCount( )
java    int functionCount( )
py      functionCount( )
php     function functionCount( )
ts      async functionCount( ): Promise<number>
es      async functionCount( )
```

Returns :
the number of functions on the module

On failure, throws an exception or returns a negative error code.

module→functionId() YModule

Retrieves the hardware identifier of the *n*th function on the module.

```
js      function functionId( functionIndex)
cpp     string functionId( int functionIndex)
m       -(NSString*) functionId : (int) functionIndex
pas     string functionId( functionIndex: integer): string
vb      function functionId( ByVal functionIndex As Integer) As String
cs      string functionId( int functionIndex)
java    String functionId( int functionIndex)
py      functionId( functionIndex)
php     function functionId( $functionIndex)
ts      async functionId( functionIndex: number): Promise<string>
es      async functionId( functionIndex)
```

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a string corresponding to the unambiguous hardware identifier of the requested module function

On failure, throws an exception or returns an empty string.

module→functionName() YModule

Retrieves the logical name of the *n*th function on the module.

js	function **functionName**(**functionIndex**)
cpp	string **functionName**(int **functionIndex**)
m	-(NSString*) **functionName** : (int) **functionIndex**
pas	string **functionName**(**functionIndex**: integer): string
vb	function **functionName**(ByVal **functionIndex** As Integer) As String
cs	string **functionName**(int **functionIndex**)
java	String **functionName**(int **functionIndex**)
py	**functionName**(**functionIndex**)
php	function **functionName**(**$functionIndex**)
ts	async **functionName**(**functionIndex**: number): Promise<string>
es	async **functionName**(**functionIndex**)

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a string corresponding to the logical name of the requested module function

On failure, throws an exception or returns an empty string.

module→functionType() YModule

Retrieves the type of the *n*th function on the module.

```
js      function functionType( functionIndex)
cpp     string functionType( int functionIndex)
pas     string functionType( functionIndex: integer): string
vb      function functionType( ByVal functionIndex As Integer) As String
cs      string functionType( int functionIndex)
java    String functionType( int functionIndex)
py      functionType( functionIndex)
php     function functionType( $functionIndex)
ts      async functionType( functionIndex: number): Promise<string>
es      async functionType( functionIndex)
```

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a string corresponding to the type of the function

On failure, throws an exception or returns an empty string.

module→functionValue() YModule

Retrieves the advertised value of the *n*th function on the module.

js	function **functionValue**(**functionIndex**)
cpp	string **functionValue**(int **functionIndex**)
m	-(NSString*) **functionValue** : (int) **functionIndex**
pas	string **functionValue**(**functionIndex**: integer): string
vb	function **functionValue**(ByVal **functionIndex** As Integer) As String
cs	string **functionValue**(int **functionIndex**)
java	String **functionValue**(int **functionIndex**)
py	**functionValue**(**functionIndex**)
php	function **functionValue**(**$functionIndex**)
ts	async **functionValue**(**functionIndex**: number): Promise<string>
es	async **functionValue**(**functionIndex**)

Parameters :

functionIndex the index of the function for which the information is desired, starting at 0 for the first function.

Returns :

a short string (up to 6 characters) corresponding to the advertised value of the requested module function

On failure, throws an exception or returns an empty string.

module→get_allSettings()
module→allSettings()

YModule

Returns all the settings and uploaded files of the module.

```
js      function get_allSettings( )
cpp     string get_allSettings( )
m       -(NSMutableData*) allSettings
pas     TByteArray get_allSettings( ): TByteArray
vb      function get_allSettings( ) As Byte
cs      byte[] get_allSettings( )
java    byte[] get_allSettings( )
uwp     async Task<byte[]> get_allSettings( )
py      get_allSettings( )
php     function get_allSettings( )
ts      async get_allSettings( ): Promise<Uint8Array>
es      async get_allSettings( )
dnp     byte[] get_allSettings( )
cp      string get_allSettings( )
cmd     YModule target get_allSettings
```

Useful to backup all the logical names, calibrations parameters, and uploaded files of a device.

Returns :
a binary buffer with all the settings.

On failure, throws an exception or returns an binary object of size 0.

module→get_beacon()
module→beacon()

YModule

Returns the state of the localization beacon.

```
js    function get_beacon( )
cpp   Y_BEACON_enum get_beacon( )
m     -(Y_BEACON_enum) beacon
pas   Integer get_beacon( ): Integer
vb    function get_beacon( ) As Integer
cs    int get_beacon( )
java  int get_beacon( )
uwp   async Task<int> get_beacon( )
py    get_beacon( )
php   function get_beacon( )
ts    async get_beacon( ): Promise<YModule_Beacon>
es    async get_beacon( )
dnp   int get_beacon( )
cp    int get_beacon( )
cmd   YModule target get_beacon
```

Returns :

either `YModule.BEACON_OFF` or `YModule.BEACON_ON`, according to the state of the localization beacon

On failure, throws an exception or returns `YModule.BEACON_INVALID`.

module→get_errorMessage()
module→errorMessage()

YModule

Returns the error message of the latest error with this module object.

```
js      function get_errorMessage( )
cpp     string get_errorMessage( )
m       -(NSString*) errorMessage
pas     string get_errorMessage( ): string
vb      function get_errorMessage( ) As String
cs      string get_errorMessage( )
java    String get_errorMessage( )
py      get_errorMessage( )
php     function get_errorMessage( )
ts      get_errorMessage( ): string
es      get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occurred while using this module object

module→get_errorType() YModule
module→errorType()

Returns the numerical error code of the latest error with this module object.

js	function **get_errorType**( )
cpp	YRETCODE **get_errorType**( )
m	-(YRETCODE) errorType
pas	YRETCODE **get_errorType**( ): YRETCODE
vb	function **get_errorType**( ) As YRETCODE
cs	YRETCODE **get_errorType**( )
java	int **get_errorType**( )
py	**get_errorType**( )
php	function **get_errorType**( )
ts	**get_errorType**( ): number
es	**get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using this module object

module→get_firmwareRelease() YModule
module→firmwareRelease()

Returns the version of the firmware embedded in the module.

```
js      function get_firmwareRelease( )
cpp     string get_firmwareRelease( )
m       -(NSString*) firmwareRelease
pas     string get_firmwareRelease( ): string
vb      function get_firmwareRelease( ) As String
cs      string get_firmwareRelease( )
java    String get_firmwareRelease( )
uwp     async Task<string> get_firmwareRelease( )
py      get_firmwareRelease( )
php     function get_firmwareRelease( )
ts      async get_firmwareRelease( ): Promise<string>
es      async get_firmwareRelease( )
dnp     string get_firmwareRelease( )
cp      string get_firmwareRelease( )
cmd     YModule target get_firmwareRelease
```

Returns :
a string corresponding to the version of the firmware embedded in the module

On failure, throws an exception or returns `YModule.FIRMWARERELEASE_INVALID`.

module→get_functionIds() YModule
module→functionIds()

Retrieve all hardware identifier that match the type passed in argument.

```
js      function get_functionIds( funType)
cpp     vector<string> get_functionIds( string funType)
m       -(NSMutableArray*) functionIds : (NSString*) funType
pas     TStringArray get_functionIds( funType: string): TStringArray
vb      function get_functionIds( ByVal funType As String) As List
cs      List<string> get_functionIds( string funType)
java    ArrayList<String> get_functionIds( String funType)
uwp     async Task<List<string>> get_functionIds( string funType)
py      get_functionIds( funType)
php     function get_functionIds( $funType)
ts      async get_functionIds( funType: string): Promise<string[]
es      async get_functionIds( funType)
dnp     string[] get_functionIds( string funType)
cp      vector<string> get_functionIds( string funType)
cmd     YModule target get_functionIds funType
```

Parameters :

funType The type of function (Relay, LightSensor, Voltage,...)

Returns :

an array of strings.

module→get_hardwareId()
module→hardwareId()

YModule

Returns the unique hardware identifier of the module.

```
js      function get_hardwareId( )
cpp     string get_hardwareId( )
m       -(NSString*) hardwareId
vb      function get_hardwareId( ) As String
cs      string get_hardwareId( )
java    String get_hardwareId( )
py      get_hardwareId( )
php     function get_hardwareId( )
ts      async get_hardwareId( ): Promise<string>
es      async get_hardwareId( )
dnp     string get_hardwareId( )
cp      string get_hardwareId( )
pas     string get_hardwareId( ): string
uwp     async Task<string> get_hardwareId( )
cmd     YModule target get_hardwareId
```

The unique hardware identifier is made of the device serial number followed by string ".module".

Returns :
a string that uniquely identifies the module

module→get_icon2d() YModule
module→icon2d()

Returns the icon of the module.

```
js      function get_icon2d( )
cpp     string get_icon2d( )
m       -(NSMutableData*) icon2d
pas     TByteArray get_icon2d( ): TByteArray
vb      function get_icon2d( ) As Byte
cs      byte[] get_icon2d( )
java    byte[] get_icon2d( )
uwp     async Task<byte[]> get_icon2d( )
py      get_icon2d( )
php     function get_icon2d( )
ts      async get_icon2d( ): Promise<Uint8Array>
es      async get_icon2d( )
dnp     byte[] get_icon2d( )
cp      string get_icon2d( )
cmd     YModule target get_icon2d
```

The icon is a PNG image and does not exceeds 1536 bytes.

Returns :

a binary buffer with module icon, in png format. On failure, throws an exception or returns `YAPI_INVALID_STRING`.

module→get_lastLogs()
module→lastLogs()

YModule

Returns a string with last logs of the module.

```
js      function get_lastLogs( )
cpp     string get_lastLogs( )
m       -(NSString*) lastLogs
pas     string get_lastLogs( ): string
vb      function get_lastLogs( ) As String
cs      string get_lastLogs( )
java    String get_lastLogs( )
uwp     async Task<string> get_lastLogs( )
py      get_lastLogs( )
php     function get_lastLogs( )
ts      async get_lastLogs( ): Promise<string>
es      async get_lastLogs( )
dnp     string get_lastLogs( )
cp      string get_lastLogs( )
cmd     YModule target get_lastLogs
```

This method return only logs that are still in the module.

Returns :

a string with last logs of the module. On failure, throws an exception or returns YAPI_INVALID_STRING.

module→get_logicalName()
module→logicalName()

YModule

Returns the logical name of the module.

js	function **get_logicalName**( )
cpp	string **get_logicalName**( )
m	-(NSString*) logicalName
pas	string **get_logicalName**( ): string
vb	function **get_logicalName**( ) As String
cs	string **get_logicalName**( )
java	String **get_logicalName**( )
uwp	async Task<string> **get_logicalName**( )
py	**get_logicalName**( )
php	function **get_logicalName**( )
ts	async **get_logicalName**( ): Promise<string>
es	async **get_logicalName**( )
dnp	string **get_logicalName**( )
cp	string **get_logicalName**( )
cmd	YModule **target get_logicalName**

Returns :
a string corresponding to the logical name of the module

On failure, throws an exception or returns `YModule.LOGICALNAME_INVALID`.

module→get_luminosity()
module→luminosity()

YModule

Returns the luminosity of the module informative LEDs (from 0 to 100).

```
js      function get_luminosity( )
cpp     int get_luminosity( )
m       -(int) luminosity
pas     LongInt get_luminosity( ): LongInt
vb      function get_luminosity( ) As Integer
cs      int get_luminosity( )
java    int get_luminosity( )
uwp     async Task<int> get_luminosity( )
py      get_luminosity( )
php     function get_luminosity( )
ts      async get_luminosity( ): Promise<number>
es      async get_luminosity( )
dnp     int get_luminosity( )
cp      int get_luminosity( )
cmd     YModule target get_luminosity
```

Returns :

an integer corresponding to the luminosity of the module informative LEDs (from 0 to 100)

On failure, throws an exception or returns `YModule.LUMINOSITY_INVALID`.

module→get_parentHub() YModule
module→parentHub()

Returns the serial number of the YoctoHub on which this module is connected.

js	function **get_parentHub**( )
cpp	string **get_parentHub**( )
m	-(NSString*) parentHub
pas	string **get_parentHub**( ): string
vb	function **get_parentHub**( ) As String
cs	string **get_parentHub**( )
java	String **get_parentHub**( )
uwp	async Task<string> **get_parentHub**( )
py	**get_parentHub**( )
php	function **get_parentHub**( )
ts	async **get_parentHub**( ): Promise<string>
es	async **get_parentHub**( )
dnp	string **get_parentHub**( )
cp	string **get_parentHub**( )
cmd	YModule **target get_parentHub**

If the module is connected by USB, or if the module is the root YoctoHub, an empty string is returned.

Returns :
a string with the serial number of the YoctoHub or an empty string

module→get_persistentSettings()
module→persistentSettings()

YModule

Returns the current state of persistent module settings.

js	function **get_persistentSettings**( )
cpp	Y_PERSISTENTSETTINGS_enum **get_persistentSettings**( )
m	-(Y_PERSISTENTSETTINGS_enum) persistentSettings
pas	Integer **get_persistentSettings**( ): Integer
vb	function **get_persistentSettings**( ) As Integer
cs	int **get_persistentSettings**( )
java	int **get_persistentSettings**( )
uwp	async Task<int> **get_persistentSettings**( )
py	**get_persistentSettings**( )
php	function **get_persistentSettings**( )
ts	async **get_persistentSettings**( ): Promise<YModule_PersistentSettings>
es	async **get_persistentSettings**( )
dnp	int **get_persistentSettings**( )
cp	int **get_persistentSettings**( )
cmd	YModule **target get_persistentSettings**

Returns :

a value among `YModule.PERSISTENTSETTINGS_LOADED`, `YModule.PERSISTENTSETTINGS_SAVED` and `YModule.PERSISTENTSETTINGS_MODIFIED` corresponding to the current state of persistent module settings

On failure, throws an exception or returns `YModule.PERSISTENTSETTINGS_INVALID`.

module→get_productId()
module→productId()

YModule

Returns the USB device identifier of the module.

```
js      function get_productId( )
cpp     int get_productId( )
m       -(int) productId
pas     LongInt get_productId( ): LongInt
vb      function get_productId( ) As Integer
cs      int get_productId( )
java    int get_productId( )
uwp     async Task<int> get_productId( )
py      get_productId( )
php     function get_productId( )
ts      async get_productId( ): Promise<number>
es      async get_productId( )
dnp     int get_productId( )
cp      int get_productId( )
cmd     YModule target get_productId
```

Returns :

an integer corresponding to the USB device identifier of the module

On failure, throws an exception or returns `YModule.PRODUCTID_INVALID`.

module→get_productName()
module→productName()

YModule

Returns the commercial name of the module, as set by the factory.

```
js      function get_productName( )
cpp     string get_productName( )
m       -(NSString*) productName
pas     string get_productName( ): string
vb      function get_productName( ) As String
cs      string get_productName( )
java    String get_productName( )
uwp     async Task<string> get_productName( )
py      get_productName( )
php     function get_productName( )
ts      async get_productName( ): Promise<string>
es      async get_productName( )
dnp     string get_productName( )
cp      string get_productName( )
cmd     YModule target get_productName
```

Returns :

a string corresponding to the commercial name of the module, as set by the factory

On failure, throws an exception or returns `YModule.PRODUCTNAME_INVALID`.

module→get_productRelease() YModule
module→productRelease()

Returns the release number of the module hardware, preprogrammed at the factory.

js	function **get_productRelease**( )
cpp	int **get_productRelease**( )
m	-(int) productRelease
pas	LongInt **get_productRelease**( ): LongInt
vb	function **get_productRelease**( ) As Integer
cs	int **get_productRelease**( )
java	int **get_productRelease**( )
uwp	async Task<int> **get_productRelease**( )
py	**get_productRelease**( )
php	function **get_productRelease**( )
ts	async **get_productRelease**( ): Promise<number>
es	async **get_productRelease**( )
dnp	int **get_productRelease**( )
cp	int **get_productRelease**( )
cmd	YModule **target get_productRelease**

The original hardware release returns value 1, revision B returns value 2, etc.

Returns :
an integer corresponding to the release number of the module hardware, preprogrammed at the factory

On failure, throws an exception or returns `YModule.PRODUCTRELEASE_INVALID`.

module→get_rebootCountdown() YModule
module→rebootCountdown()

Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

js	function **get_rebootCountdown**( )
cpp	int **get_rebootCountdown**( )
m	-(int) rebootCountdown
pas	LongInt **get_rebootCountdown**( ): LongInt
vb	function **get_rebootCountdown**( ) As Integer
cs	int **get_rebootCountdown**( )
java	int **get_rebootCountdown**( )
uwp	async Task<int> **get_rebootCountdown**( )
py	**get_rebootCountdown**( )
php	function **get_rebootCountdown**( )
ts	async **get_rebootCountdown**( ): Promise<number>
es	async **get_rebootCountdown**( )
dnp	int **get_rebootCountdown**( )
cp	int **get_rebootCountdown**( )
cmd	YModule **target get_rebootCountdown**

Returns :

an integer corresponding to the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled

On failure, throws an exception or returns `YModule.REBOOTCOUNTDOWN_INVALID`.

module→get_serialNumber() YModule
module→serialNumber()

Returns the serial number of the module, as set by the factory.

js	function **get_serialNumber**( )
cpp	string **get_serialNumber**( )
m	-(NSString*) serialNumber
pas	string **get_serialNumber**( ): string
vb	function **get_serialNumber**( ) As String
cs	string **get_serialNumber**( )
java	String **get_serialNumber**( )
uwp	async Task<string> **get_serialNumber**( )
py	**get_serialNumber**( )
php	function **get_serialNumber**( )
ts	async **get_serialNumber**( ): Promise<string>
es	async **get_serialNumber**( )
dnp	string **get_serialNumber**( )
cp	string **get_serialNumber**( )
cmd	YModule **target get_serialNumber**

Returns :

a string corresponding to the serial number of the module, as set by the factory

On failure, throws an exception or returns `YModule.SERIALNUMBER_INVALID`.

module→get_subDevices()
module→subDevices()

YModule

Returns a list of all the modules that are plugged into the current module.

```
js      function get_subDevices( )
cpp     vector<string> get_subDevices( )
m       -(NSMutableArray*) subDevices
pas     TStringArray get_subDevices( ): TStringArray
vb      function get_subDevices( ) As List
cs      List<string> get_subDevices( )
java    ArrayList<String> get_subDevices( )
uwp     async Task<List<string>> get_subDevices( )
py      get_subDevices( )
php     function get_subDevices( )
ts      async get_subDevices( ): Promise<string[]
es      async get_subDevices( )
dnp     string[] get_subDevices( )
cp      vector<string> get_subDevices( )
cmd     YModule target get_subDevices
```

This method only makes sense when called for a YoctoHub/VirtualHub. Otherwise, an empty array will be returned.

Returns :
an array of strings containing the sub modules.

module→get_upTime() YModule
module→upTime()

Returns the number of milliseconds spent since the module was powered on.

js	function **get_upTime**( )
cpp	s64 **get_upTime**( )
m	-(s64) upTime
pas	int64 **get_upTime**( ): int64
vb	function **get_upTime**( ) As Long
cs	long **get_upTime**( )
java	long **get_upTime**( )
uwp	async Task<long> **get_upTime**( )
py	**get_upTime**( )
php	function **get_upTime**( )
ts	async **get_upTime**( ): Promise<number>
es	async **get_upTime**( )
dnp	long **get_upTime**( )
cp	s64 **get_upTime**( )
cmd	YModule **target get_upTime**

Returns :
an integer corresponding to the number of milliseconds spent since the module was powered on

On failure, throws an exception or returns `YModule.UPTIME_INVALID`.

module→get_url()
module→url()

YModule

Returns the URL used to access the module.

```
js      function get_url( )
cpp     string get_url( )
m       -(NSString*) url
pas     string get_url( ): string
vb      function get_url( ) As String
cs      string get_url( )
java    String get_url( )
uwp     async Task<string> get_url( )
py      get_url( )
php     function get_url( )
ts      async get_url( ): Promise<string>
es      async get_url( )
dnp     string get_url( )
cp      string get_url( )
cmd     YModule target get_url
```

If the module is connected by USB, the string 'usb' is returned.

Returns :
a string with the URL of the module.

module→get_usbCurrent()
module→usbCurrent()

YModule

Returns the current consumed by the module on the USB bus, in milli-amps.

```
js      function get_usbCurrent( )
cpp     int get_usbCurrent( )
m       -(int) usbCurrent
pas     LongInt get_usbCurrent( ): LongInt
vb      function get_usbCurrent( ) As Integer
cs      int get_usbCurrent( )
java    int get_usbCurrent( )
uwp     async Task<int> get_usbCurrent( )
py      get_usbCurrent( )
php     function get_usbCurrent( )
ts      async get_usbCurrent( ): Promise<number>
es      async get_usbCurrent( )
dnp     int get_usbCurrent( )
cp      int get_usbCurrent( )
cmd     YModule target get_usbCurrent
```

Returns :

an integer corresponding to the current consumed by the module on the USB bus, in milli-amps

On failure, throws an exception or returns `YModule.USBCURRENT_INVALID`.

module→get_userData() YModule
module→userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

```
js      function get_userData( )
cpp     void * get_userData( )
m       -(id) userData
pas     Tobject get_userData( ): Tobject
vb      function get_userData( ) As Object
cs      object get_userData( )
java    Object get_userData( )
py      get_userData( )
php     function get_userData( )
ts      async get_userData( ): Promise<object|null>
es      async get_userData( )
```

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

module→get_userVar()
module→userVar()

YModule

Returns the value previously stored in this attribute.

js	function **get_userVar**( )
cpp	int **get_userVar**( )
m	-(int) userVar
pas	LongInt **get_userVar**( ): LongInt
vb	function **get_userVar**( ) As Integer
cs	int **get_userVar**( )
java	int **get_userVar**( )
uwp	async Task<int> **get_userVar**( )
py	**get_userVar**( )
php	function **get_userVar**( )
ts	async **get_userVar**( ): Promise<number>
es	async **get_userVar**( )
dnp	int **get_userVar**( )
cp	int **get_userVar**( )
cmd	YModule **target get_userVar**

On startup and after a device reboot, the value is always reset to zero.

Returns :
an integer corresponding to the value previously stored in this attribute

On failure, throws an exception or returns `YModule.USERVAR_INVALID`.

module→hasFunction() YModule

Tests if the device includes a specific function.

```
js      function hasFunction( funcId)
cpp     bool hasFunction( string funcId)
m       -(bool) hasFunction : (NSString*) funcId
pas     boolean hasFunction( funcId: string): boolean
vb      function hasFunction( ByVal funcId As String) As Boolean
cs      bool hasFunction( string funcId)
java    boolean hasFunction( String funcId)
uwp     async Task<bool> hasFunction( string funcId)
py      hasFunction( funcId)
php     function hasFunction( $funcId)
ts      async hasFunction( funcId: string): Promise<boolean>
es      async hasFunction( funcId)
dnp     bool hasFunction( string funcId)
cp      bool hasFunction( string funcId)
cmd     YModule target hasFunction funcId
```

This method takes a function identifier and returns a boolean.

Parameters :

funcId the requested function identifier

Returns :

true if the device has the function identifier

module→isOnline() YModule

Checks if the module is currently reachable, without raising any error.

js	function **isOnline**( )
cpp	bool **isOnline**( )
m	-(BOOL) **isOnline**
pas	boolean **isOnline**( ): boolean
vb	function **isOnline**( ) As Boolean
cs	bool **isOnline**( )
java	boolean **isOnline**( )
py	**isOnline**( )
php	function **isOnline**( )
ts	async **isOnline**( ): Promise<boolean>
es	async **isOnline**( )
dnp	bool **isOnline**( )
cp	bool **isOnline**( )

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

Returns :
`true` if the module can be reached, and `false` otherwise

module→isOnline_async() YModule

Checks if the module is currently reachable, without raising any error.

```
js  function isOnline_async( callback, context)
```

If there are valid cached values for the module, that have not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the requested module.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking Firefox JavaScript VM that does not implement context switching during blocking I/O calls.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving module object and the boolean result

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

module→load() YModule

Preloads the module cache with a specified validity duration.

```
js      function load( msValidity)
cpp     YRETCODE load( int msValidity)
m       -(YRETCODE) load : (u64) msValidity
pas     YRETCODE load( msValidity: u64): YRETCODE
vb      function load( ByVal msValidity As Long) As YRETCODE
cs      YRETCODE load( ulong msValidity)
java    int load( long msValidity)
py      load( msValidity)
php     function load( $msValidity)
ts      async load( msValidity: number): Promise<number>
es      async load( msValidity)
```

By default, whenever accessing a device, all module attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded module parameters, in milliseconds

Returns :

YAPI.SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→load_async() YModule

Preloads the module cache with a specified validity duration (asynchronous version).

```
js  function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all module attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking Firefox JavaScript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous JavaScript calls for more details.

Parameters :

msValidity	an integer corresponding to the validity of the loaded module parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving module object and the error code (or YAPI.SUCCESS)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

module→log() YModule

Adds a text message to the device logs.

```
js      function log( text)
cpp     int log( string text)
m       -(int) log : (NSString*) text
pas     LongInt log( text: string): LongInt
vb      function log( ByVal text As String) As Integer
cs      int log( string text)
java    int log( String text)
uwp     async Task<int> log( string text)
py      log( text)
php     function log( $text)
ts      async log( text: string): Promise<number>
es      async log( text)
dnp     int log( string text)
cp      int log( string text)
cmd     YModule target log text
```

This function is useful in particular to trace the execution of HTTP callbacks. If a newline is desired after the message, it must be included in the string.

Parameters :

text the string to append to the logs.

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→nextModule() YModule

Continues the module enumeration started using `yFirstModule()`.

js	function **nextModule**( )
cpp	YModule * **nextModule**( )
m	-(nullable YModule*) **nextModule**
pas	TYModule **nextModule**( ): TYModule
vb	function **nextModule**( ) As YModule
cs	YModule **nextModule**( )
java	YModule **nextModule**( )
uwp	YModule **nextModule**( )
py	**nextModule**( )
php	function **nextModule**( )
ts	**nextModule**( ): YModule | null
es	**nextModule**( )

Caution: You can't make any assumption about the returned modules order. If you want to find a specific module, use `Module.findModule()` and a hardwareID or a logical name.

Returns :

a pointer to a `YModule` object, corresponding to the next module found, or a `null` pointer if there are no more modules to enumerate.

module→reboot() YModule

Schedules a simple module reboot after the given number of seconds.

```
js    function reboot( secBeforeReboot)
cpp   int reboot( int secBeforeReboot)
m     -(int) reboot : (int) secBeforeReboot
pas   LongInt reboot( secBeforeReboot: LongInt): LongInt
vb    function reboot( ByVal secBeforeReboot As Integer) As Integer
cs    int reboot( int secBeforeReboot)
java  int reboot( int secBeforeReboot)
uwp   async Task<int> reboot( int secBeforeReboot)
py    reboot( secBeforeReboot)
php   function reboot( $secBeforeReboot)
ts    async reboot( secBeforeReboot: number): Promise<number>
es    async reboot( secBeforeReboot)
dnp   int reboot( int secBeforeReboot)
cp    int reboot( int secBeforeReboot)
cmd   YModule target reboot secBeforeReboot
```

Parameters :

secBeforeReboot number of seconds before rebooting

Returns :

YAPI.SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→registerBeaconCallback() YModule

Register a callback function, to be called when the localization beacon of the module has been changed.

```
js    function registerBeaconCallback( callback)
cpp   int registerBeaconCallback( YModuleBeaconCallback callback)
m     -(int) registerBeaconCallback : (YModuleBeaconCallback _Nullable) callback
pas   LongInt registerBeaconCallback( callback: TYModuleBeaconCallback): LongInt
vb    function registerBeaconCallback( ByVal callback As YModuleBeaconCallback) As Integer
cs    int registerBeaconCallback( BeaconCallback callback)
java  int registerBeaconCallback( BeaconCallback callback)
uwp   async Task<int> registerBeaconCallback( BeaconCallback callback)
py    registerBeaconCallback( callback)
php   function registerBeaconCallback( $callback)
ts    async registerBeaconCallback( callback: YModuleBeaconCallback | null): Promise<number>
es    async registerBeaconCallback( callback)
```

The callback function should take two arguments: the YModule object of which the beacon has changed, and an integer describing the new beacon state.

Parameters :

callback The callback function to call, or `null` to unregister a

module→registerConfigChangeCallback() YModule

Register a callback function, to be called when a persistent settings in a device configuration has been changed (e.g.

```
js    function registerConfigChangeCallback( callback)
cpp   int registerConfigChangeCallback( YModuleConfigChangeCallback callback)
m     -(int) registerConfigChangeCallback : (YModuleConfigChangeCallback _Nullable) callback
pas   LongInt registerConfigChangeCallback( callback: TYModuleConfigChangeCallback): LongInt
vb    function registerConfigChangeCallback( ByVal callback As YModuleConfigChangeCallback) As
                                             Integer
cs    int registerConfigChangeCallback( ConfigChangeCallback callback)
java  int registerConfigChangeCallback( ConfigChangeCallback callback)
uwp   async Task<int> registerConfigChangeCallback( ConfigChangeCallback callback)
py    registerConfigChangeCallback( callback)
php   function registerConfigChangeCallback( $callback)
ts    async registerConfigChangeCallback( callback: YModuleConfigChangeCallback | null):
                                          Promise<number>
es    async registerConfigChangeCallback( callback)
```

change of unit, etc).

Parameters :

callback a procedure taking a YModule parameter, or `null`

module→registerLogCallback() YModule

Registers a device log callback function.

```
js      function registerLogCallback( callback)
cpp     int registerLogCallback( YModuleLogCallback callback)
m       -(int) registerLogCallback : (YModuleLogCallback _Nullable) callback
pas     LongInt registerLogCallback( callback: TYModuleLogCallback): LongInt
vb      function registerLogCallback( ByVal callback As YModuleLogCallback) As Integer
cs      int registerLogCallback( LogCallback callback)
java    int registerLogCallback( LogCallback callback)
uwp     async Task<int> registerLogCallback( LogCallback callback)
py      registerLogCallback( callback)
php     function registerLogCallback( $callback)
ts      async registerLogCallback( callback: YModuleLogCallback | null): Promise<number>
es      async registerLogCallback( callback)
```

This callback will be called each time that a module sends a new log message. Mostly useful to debug a Yoctopuce module.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the module object that emitted the log message, and the character string containing the log.

module→revertFromFlash() YModule

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

js	function **revertFromFlash**( )
cpp	int **revertFromFlash**( )
m	-(int) **revertFromFlash**
pas	LongInt **revertFromFlash**( ): LongInt
vb	function **revertFromFlash**( ) As Integer
cs	int **revertFromFlash**( )
java	int **revertFromFlash**( )
uwp	async Task<int> **revertFromFlash**( )
py	**revertFromFlash**( )
php	function **revertFromFlash**( )
ts	async **revertFromFlash**( ): Promise<number>
es	async **revertFromFlash**( )
dnp	int **revertFromFlash**( )
cp	int **revertFromFlash**( )
cmd	YModule **target revertFromFlash**

Returns :
YAPI.SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→saveToFlash() YModule

Saves current settings in the nonvolatile memory of the module.

```
js      function saveToFlash( )
cpp     int saveToFlash( )
m       -(int) saveToFlash
pas     LongInt saveToFlash( ): LongInt
vb      function saveToFlash( ) As Integer
cs      int saveToFlash( )
java    int saveToFlash( )
uwp     async Task<int> saveToFlash( )
py      saveToFlash( )
php     function saveToFlash( )
ts      async saveToFlash( ): Promise<number>
es      async saveToFlash( )
dnp     int saveToFlash( )
cp      int saveToFlash( )
cmd     YModule target saveToFlash
```

Warning: the number of allowed save operations during a module life is limited (about 100000 cycles). Do not call this function within a loop.

Returns :
YAPI.SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_allSettings() YModule
module→setAllSettings()

Restores all the settings of the device.

```
js      function set_allSettings( settings)
cpp     int set_allSettings( string settings)
m       -(int) setAllSettings : (NSData*) settings
pas     LongInt set_allSettings( settings: TByteArray): LongInt
vb      procedure set_allSettings( ByVal settings As Byte()
cs      int set_allSettings( byte[] settings)
java    int set_allSettings( byte[] settings)
uwp     async Task<int> set_allSettings( byte[] settings)
py      set_allSettings( settings)
php     function set_allSettings( $settings)
ts      async set_allSettings( settings: Uint8Array): Promise<number>
es      async set_allSettings( settings)
dnp     int set_allSettings( byte[] settings)
cp      int set_allSettings( string settings)
cmd     YModule target set_allSettings settings
```

Useful to restore all the logical names and calibrations parameters of a module from a backup.Remember to call the `saveToFlash()` method of the module if the modifications must be kept.

Parameters :

settings a binary buffer with all the settings.

Returns :

`YAPI.SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_allSettingsAndFiles() YModule
module→setAllSettingsAndFiles()

Restores all the settings and uploaded files to the module.

```
js      function set_allSettingsAndFiles( settings)
cpp     int set_allSettingsAndFiles( string settings)
m       -(int) setAllSettingsAndFiles : (NSData*) settings
pas     LongInt set_allSettingsAndFiles( settings: TByteArray): LongInt
vb      procedure set_allSettingsAndFiles( ByVal settings As Byte()
cs      int set_allSettingsAndFiles( byte[] settings)
java    int set_allSettingsAndFiles( byte[] settings)
uwp     async Task<int> set_allSettingsAndFiles( byte[] settings)
py      set_allSettingsAndFiles( settings)
php     function set_allSettingsAndFiles( $settings)
ts      async set_allSettingsAndFiles( settings: Uint8Array): Promise<number>
es      async set_allSettingsAndFiles( settings)
dnp     int set_allSettingsAndFiles( byte[] settings)
cp      int set_allSettingsAndFiles( string settings)
cmd     YModule target set_allSettingsAndFiles settings
```

This method is useful to restore all the logical names and calibrations parameters, uploaded files etc. of a device from a backup. Remember to call the `saveToFlash()` method of the module if the modifications must be kept.

Parameters :

settings a binary buffer with all the settings.

Returns :

`YAPI.SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_beacon()
module→setBeacon()

YModule

Turns on or off the module localization beacon.

```
js      function set_beacon( newval)
cpp     int set_beacon( Y_BEACON_enum newval)
m       -(int) setBeacon : (Y_BEACON_enum) newval
pas     integer set_beacon( newval: Integer): integer
vb      function set_beacon( ByVal newval As Integer) As Integer
cs      int set_beacon( int newval)
java    int set_beacon( int newval)
uwp     async Task<int> set_beacon( int newval)
py      set_beacon( newval)
php     function set_beacon( $newval)
ts      async set_beacon( newval: YModule_Beacon): Promise<number>
es      async set_beacon( newval)
dnp     int set_beacon( int newval)
cp      int set_beacon( int newval)
cmd     YModule target set_beacon newval
```

Parameters :

newval either `YModule.BEACON_OFF` or `YModule.BEACON_ON`

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_logicalName() YModule
module→setLogicalName()

Changes the logical name of the module.

js	function **set_logicalName**(**newval**)
cpp	int **set_logicalName**(string **newval**)
m	-(int) setLogicalName : (NSString*) **newval**
pas	integer **set_logicalName**(**newval**: string): integer
vb	function **set_logicalName**(ByVal **newval** As String) As Integer
cs	int **set_logicalName**(string **newval**)
java	int **set_logicalName**(String **newval**)
uwp	async Task<int> **set_logicalName**(string **newval**)
py	**set_logicalName**(**newval**)
php	function **set_logicalName**(**$newval**)
ts	async **set_logicalName**(**newval**: string): Promise<number>
es	async **set_logicalName**(**newval**)
dnp	int **set_logicalName**(string **newval**)
cp	int **set_logicalName**(string **newval**)
cmd	YModule **target set_logicalName newval**

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the module

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_luminosity() YModule
module→setLuminosity()

Changes the luminosity of the module informative leds.

```
js      function set_luminosity( newval)
cpp     int set_luminosity( int newval)
m       -(int) setLuminosity : (int) newval
pas     integer set_luminosity( newval: LongInt): integer
vb      function set_luminosity( ByVal newval As Integer) As Integer
cs      int set_luminosity( int newval)
java    int set_luminosity( int newval)
uwp     async Task<int> set_luminosity( int newval)
py      set_luminosity( newval)
php     function set_luminosity( $newval)
ts      async set_luminosity( newval: number): Promise<number>
es      async set_luminosity( newval)
dnp     int set_luminosity( int newval)
cp      int set_luminosity( int newval)
cmd     YModule target set_luminosity newval
```

The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer corresponding to the luminosity of the module informative leds

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_userData()
module→setUserData()

YModule

Stores a user context provided as argument in the userData attribute of the function.

```
js      function set_userData( data)
cpp     void set_userData( void * data)
m       -(void) setUserData : (id) data
pas     set_userData( data: Tobject)
vb      procedure set_userData( ByVal data As Object)
cs      void set_userData( object data)
java    void set_userData( Object data)
py      set_userData( data)
php     function set_userData( $data)
ts      async set_userData( data: object|null): Promise<void>
es      async set_userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

module→set_userVar() YModule
module→setUserVar()

Stores a 32 bit value in the device RAM.

```
js      function set_userVar( newval)
cpp     int set_userVar( int newval)
m       -(int) setUserVar : (int) newval
pas     integer set_userVar( newval: LongInt): integer
vb      function set_userVar( ByVal newval As Integer) As Integer
cs      int set_userVar( int newval)
java    int set_userVar( int newval)
uwp     async Task<int> set_userVar( int newval)
py      set_userVar( newval)
php     function set_userVar( $newval)
ts      async set_userVar( newval: number): Promise<number>
es      async set_userVar( newval)
dnp     int set_userVar( int newval)
cp      int set_userVar( int newval)
cmd     YModule target set_userVar newval
```

This attribute is at programmer disposal, should he need to store a state variable. On startup and after a device reboot, the value is always reset to zero.

Parameters :

newval an integer

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→triggerConfigChangeCallback() YModule

Triggers a configuration change callback, to check if they are supported or not.

```
js      function triggerConfigChangeCallback( )
cpp     int triggerConfigChangeCallback( )
m       -(int) triggerConfigChangeCallback
pas     LongInt triggerConfigChangeCallback( ): LongInt
vb      function triggerConfigChangeCallback( ) As Integer
cs      int triggerConfigChangeCallback( )
java    int triggerConfigChangeCallback( )
uwp     async Task<int> triggerConfigChangeCallback( )
py      triggerConfigChangeCallback( )
php     function triggerConfigChangeCallback( )
ts      async triggerConfigChangeCallback( ): Promise<number>
es      async triggerConfigChangeCallback( )
dnp     int triggerConfigChangeCallback( )
cp      int triggerConfigChangeCallback( )
cmd     YModule target triggerConfigChangeCallback
```

module→triggerFirmwareUpdate() YModule

Schedules a module reboot into special firmware update mode.

```
js    function triggerFirmwareUpdate( secBeforeReboot)
cpp   int triggerFirmwareUpdate( int secBeforeReboot)
m     -(int) triggerFirmwareUpdate : (int) secBeforeReboot
pas   LongInt triggerFirmwareUpdate( secBeforeReboot: LongInt): LongInt
vb    function triggerFirmwareUpdate( ByVal secBeforeReboot As Integer) As Integer
cs    int triggerFirmwareUpdate( int secBeforeReboot)
java  int triggerFirmwareUpdate( int secBeforeReboot)
uwp   async Task<int> triggerFirmwareUpdate( int secBeforeReboot)
py    triggerFirmwareUpdate( secBeforeReboot)
php   function triggerFirmwareUpdate( $secBeforeReboot)
ts    async triggerFirmwareUpdate( secBeforeReboot: number): Promise<number>
es    async triggerFirmwareUpdate( secBeforeReboot)
dnp   int triggerFirmwareUpdate( int secBeforeReboot)
cp    int triggerFirmwareUpdate( int secBeforeReboot)
cmd   YModule target triggerFirmwareUpdate secBeforeReboot
```

Parameters :

secBeforeReboot number of seconds before rebooting

Returns :

YAPI.SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

module→updateFirmware() YModule

Prepares a firmware update of the module.

```
js      function updateFirmware( path)
cpp     YFirmwareUpdate updateFirmware( string path)
m       -(YFirmwareUpdate*) updateFirmware : (NSString*) path
pas     TYFirmwareUpdate updateFirmware( path: string): TYFirmwareUpdate
vb      function updateFirmware( ByVal path As String) As YFirmwareUpdate
cs      YFirmwareUpdate updateFirmware( string path)
java    YFirmwareUpdate updateFirmware( String path)
uwp     async Task<YFirmwareUpdate> updateFirmware( string path)
py      updateFirmware( path)
php     function updateFirmware( $path)
ts      async updateFirmware( path: string): Promise<YFirmwareUpdate>
es      async updateFirmware( path)
dnp     YFirmwareUpdateProxy updateFirmware( string path)
cp      YFirmwareUpdateProxy* updateFirmware( string path)
cmd     YModule target updateFirmware path
```

This method returns a `YFirmwareUpdate` object which handles the firmware update process.

Parameters :

path the path of the `.byn` file to use.

Returns :

a `YFirmwareUpdate` object or NULL on error.

module→updateFirmwareEx() YModule

Prepares a firmware update of the module.

```
js      function updateFirmwareEx( path, force)
cpp     YFirmwareUpdate updateFirmwareEx( string path, bool force)
m       -(YFirmwareUpdate*) updateFirmwareEx : (NSString*) path
                                             : (bool) force
pas     TYFirmwareUpdate updateFirmwareEx( path: string, force: boolean): TYFirmwareUpdate
vb      function updateFirmwareEx( ByVal path As String,
                                   ByVal force As Boolean) As YFirmwareUpdate
cs      YFirmwareUpdate updateFirmwareEx( string path, bool force)
java    YFirmwareUpdate updateFirmwareEx( String path, boolean force)
uwp     async Task<YFirmwareUpdate> updateFirmwareEx( string path, bool force)
py      updateFirmwareEx( path, force)
php     function updateFirmwareEx( $path, $force)
ts      async updateFirmwareEx( path: string, force: boolean): Promise<YFirmwareUpdate>
es      async updateFirmwareEx( path, force)
dnp     YFirmwareUpdateProxy updateFirmwareEx( string path, bool force)
cp      YFirmwareUpdateProxy* updateFirmwareEx( string path,
                                                bool force)
cmd     YModule target updateFirmwareEx path force
```

This method returns a `YFirmwareUpdate` object which handles the firmware update process.

Parameters :

path the path of the `.byn` file to use.

force true to force the firmware update even if some prerequisites appear not to be met

Returns :

a `YFirmwareUpdate` object or NULL on error.

module→wait_async() YModule

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js  function wait_async( callback, context)
ts  wait_async( callback: Function, context: object)
es  wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

24.3. Class YDisplay

Display control interface, available for instance in the Yocto-Display, the Yocto-MaxiDisplay, the Yocto-MaxiDisplay-G or the Yocto-MiniDisplay

The `YDisplay` class allows to drive Yoctopuce displays. Yoctopuce display interface has been designed to easily show information and images. The device provides built-in multi-layer rendering. Layers can be drawn offline, individually, and freely moved on the display. It can also replay recorded sequences (animations).

In order to draw on the screen, you should use the `display.get_displayLayer` method to retrieve the layer(s) on which you want to draw, and then use methods defined in `YDisplayLayer` to draw on the layers.

In order to use the functions described here, you should include:

`js`	<script type='text/javascript' src='yocto_display.js'></script>
`cpp`	#include "yocto_display.h"
`m`	#import "yocto_display.h"
`pas`	uses yocto_display;
`vb`	yocto_display.vb
`cs`	yocto_display.cs
`java`	import com.yoctopuce.YoctoAPI.YDisplay;
`uwp`	import com.yoctopuce.YoctoAPI.YDisplay;
`py`	from yocto_display import *
`php`	require_once('yocto_display.php');
`ts`	in HTML: import { YDisplay } from '../../dist/esm/yocto_display.js'; in Node.js: import { YDisplay } from 'yoctolib-cjs/yocto_display.js';
`es`	in HTML: <script src="../../lib/yocto_display.js"></script> in node.js: require('yoctolib-es2017/yocto_display.js');
`dnp`	import YoctoProxyAPI.YDisplayProxy
`cp`	#include "yocto_display_proxy.h"
`vi`	YDisplay.vi
`ml`	import YoctoProxyAPI.YDisplayProxy

Global functions

YDisplay.FindDisplay(**func**)

Retrieves a display for a given identifier.

YDisplay.FindDisplayInContext(**yctx**, **func**)

Retrieves a display for a given identifier in a YAPI context.

YDisplay.FirstDisplay()

Starts the enumeration of displays currently accessible.

YDisplay.FirstDisplayInContext(**yctx**)

Starts the enumeration of displays currently accessible.

YDisplay.GetSimilarFunctions()

Enumerates all functions of type Display available on the devices currently reachable by the library, and returns their unique hardware ID.

`YDisplay` properties

display→AdvertisedValue *[read-only]*

Short string representing the current state of the function.

display→Brightness *[writable]*

Luminosity of the module informative LEDs (from 0 to 100).

display→DisplayHeight *[read-only]*

Display height, in pixels.

display→DisplayType *[read-only]*

Display type: monochrome, gray levels or full color.

display→DisplayWidth *[read-only]*

Display width, in pixels.

display→FriendlyName *[read-only]*

Global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

display→FunctionId *[read-only]*

Hardware identifier of the display, without reference to the module.

display→HardwareId *[read-only]*

Unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`.

display→IsOnline *[read-only]*

Checks if the function is currently reachable.

display→LayerCount *[read-only]*

Number of available layers to draw on.

display→LayerHeight *[read-only]*

Height of the layers to draw on, in pixels.

display→LayerWidth *[read-only]*

Width of the layers to draw on, in pixels.

display→LogicalName *[writable]*

Logical name of the function.

display→Orientation *[writable]*

Currently selected display orientation.

display→SerialNumber *[read-only]*

Serial number of the module, as set by the factory.

display→StartupSeq *[writable]*

Name of the sequence to play when the displayed is powered on.

`YDisplay` methods

display→clearCache()

Invalidates the cache.

display→copyLayerContent(**srcLayerId**, **dstLayerId**)

Copies the whole content of a layer to another layer.

display→describe()

Returns a short text that describes unambiguously the instance of the display in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

display→fade(**brightness**, **duration**)

Smoothly changes the brightness of the screen to produce a fade-in or fade-out effect.

display→get_advertisedValue()

Returns the current value of the display (no more than 6 characters).

display→get_brightness()

Returns the luminosity of the module informative LEDs (from 0 to 100).

display→get_displayHeight()

Returns the display height, in pixels.

display→get_displayLayer(layerId)
Returns a YDisplayLayer object that can be used to draw on the specified layer.

display→get_displayType()
Returns the display type: monochrome, gray levels or full color.

display→get_displayWidth()
Returns the display width, in pixels.

display→get_enabled()
Returns true if the screen is powered, false otherwise.

display→get_errorMessage()
Returns the error message of the latest error with the display.

display→get_errorType()
Returns the numerical error code of the latest error with the display.

display→get_friendlyName()
Returns a global identifier of the display in the format `MODULE_NAME.FUNCTION_NAME`.

display→get_functionDescriptor()
Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

display→get_functionId()
Returns the hardware identifier of the display, without reference to the module.

display→get_hardwareId()
Returns the unique hardware identifier of the display in the form `SERIAL.FUNCTIONID`.

display→get_layerCount()
Returns the number of available layers to draw on.

display→get_layerHeight()
Returns the height of the layers to draw on, in pixels.

display→get_layerWidth()
Returns the width of the layers to draw on, in pixels.

display→get_logicalName()
Returns the logical name of the display.

display→get_module()
Gets the `YModule` object for the device on which the function is located.

display→get_module_async(callback, context)
Gets the `YModule` object for the device on which the function is located (asynchronous version).

display→get_orientation()
Returns the currently selected display orientation.

display→get_serialNumber()
Returns the serial number of the module, as set by the factory.

display→get_startupSeq()
Returns the name of the sequence to play when the displayed is powered on.

display→get_userData()
Returns the value of the userData attribute, as previously stored using method `set_userData`.

display→isOnline()
Checks if the display is currently reachable, without raising any error.

display→isOnline_async(callback, context)
Checks if the display is currently reachable, without raising any error (asynchronous version).

display→isReadOnly()

Test if the function is readOnly.

display→load(**msValidity**)

Preloads the display cache with a specified validity duration.

display→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.

display→load_async(**msValidity**, **callback**, **context**)

Preloads the display cache with a specified validity duration (asynchronous version).

display→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

display→newSequence()

Starts to record all display commands into a sequence, for later replay.

display→nextDisplay()

Continues the enumeration of displays started using `yFirstDisplay()`.

display→pauseSequence(**delay_ms**)

Waits for a specified delay (in milliseconds) before playing next commands in current sequence.

display→playSequence(**sequenceName**)

Replays a display sequence previously recorded using `newSequence()` and `saveSequence()`.

display→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

display→resetAll()

Clears the display screen and resets all display layers to their default state.

display→saveSequence(**sequenceName**)

Stops recording display commands and saves the sequence into the specified file on the display internal memory.

display→set_brightness(**newval**)

Changes the brightness of the display.

display→set_enabled(**newval**)

Changes the power state of the display.

display→set_logicalName(**newval**)

Changes the logical name of the display.

display→set_orientation(**newval**)

Changes the display orientation.

display→set_startupSeq(**newval**)

Changes the name of the sequence to play when the displayed is powered on.

display→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

display→stopSequence()

Stops immediately any ongoing sequence replay.

display→swapLayerContent(**layerIdA**, **layerIdB**)

Swaps the whole content of two layers.

display→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

display→upload(**pathname**, **content**)

Uploads an arbitrary file (for instance a GIF file) to the display, to the specified full path name.

display→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YDisplay.FindDisplay() YDisplay YDisplay.FindDisplay()

Retrieves a display for a given identifier.

js	function **yFindDisplay**(**func**)
cpp	YDisplay* **FindDisplay**(string **func**)
m	+(YDisplay*) **FindDisplay** : (NSString*) **func**
pas	TYDisplay **yFindDisplay**(**func**: string): TYDisplay
vb	function **FindDisplay**(ByVal **func** As String) As YDisplay
cs	static YDisplay **FindDisplay**(string **func**)
java	static YDisplay **FindDisplay**(String **func**)
uwp	static YDisplay **FindDisplay**(string **func**)
py	**FindDisplay**(**func**)
php	function **FindDisplay**(**$func**)
ts	static **FindDisplay**(**func**: string): YDisplay
es	static **FindDisplay**(**func**)
dnp	static YDisplayProxy **FindDisplay**(string **func**)
cp	static YDisplayProxy * **FindDisplay**(string **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the display is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDisplay.isOnline()` to test if the display is indeed online at a given time. In case of ambiguity when looking for a display by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the display, for instance `YD128X32.display`.

Returns :

a `YDisplay` object allowing you to drive the display.

YDisplay.FindDisplayInContext() YDisplay
YDisplay.FindDisplayInContext()

Retrieves a display for a given identifier in a YAPI context.

```
java  static YDisplay FindDisplayInContext( YAPIContext yctx, String func)
uwp   static YDisplay FindDisplayInContext( YAPIContext yctx, string func)
ts    static FindDisplayInContext( yctx: YAPIContext, func: string): YDisplay
es    static FindDisplayInContext( yctx, func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the display is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YDisplay.isOnline()` to test if the display is indeed online at a given time. In case of ambiguity when looking for a display by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the display, for instance `YD128X32.display`.

Returns :

a `YDisplay` object allowing you to drive the display.

YDisplay.FirstDisplay() YDisplay
YDisplay.FirstDisplay()

Starts the enumeration of displays currently accessible.

js	function **yFirstDisplay**( )
cpp	YDisplay * **FirstDisplay**( )
m	+(YDisplay*) **FirstDisplay**
pas	TYDisplay **yFirstDisplay**( ): TYDisplay
vb	function **FirstDisplay**( ) As YDisplay
cs	static YDisplay **FirstDisplay**( )
java	static YDisplay **FirstDisplay**( )
uwp	static YDisplay **FirstDisplay**( )
py	**FirstDisplay**( )
php	function **FirstDisplay**( )
ts	static **FirstDisplay**( ): YDisplay | null
es	static **FirstDisplay**( )

Use the method `YDisplay.nextDisplay()` to iterate on next displays.

Returns :

a pointer to a `YDisplay` object, corresponding to the first display currently online, or a `null` pointer if there are none.

YDisplay.FirstDisplayInContext() YDisplay
YDisplay.FirstDisplayInContext()

Starts the enumeration of displays currently accessible.

java	static YDisplay **FirstDisplayInContext**(YAPIContext **yctx**)
uwp	static YDisplay **FirstDisplayInContext**(YAPIContext **yctx**)
ts	static **FirstDisplayInContext**(**yctx**: YAPIContext): YDisplay | null
es	static **FirstDisplayInContext**(**yctx**)

Use the method `YDisplay.nextDisplay()` to iterate on next displays.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YDisplay` object, corresponding to the first display currently online, or a `null` pointer if there are none.

YDisplay.GetSimilarFunctions() YDisplay
YDisplay.GetSimilarFunctions()

Enumerates all functions of type Display available on the devices currently reachable by the library, and returns their unique hardware ID.

dnp static new string[] **GetSimilarFunctions**( )

cp static vector<string> **GetSimilarFunctions**( )

Each of these IDs can be provided as argument to the method `YDisplay.FindDisplay` to obtain an object that can control the corresponding device.

Returns :

an array of strings, each string containing the unique hardwareId of a device function currently connected.

display→AdvertisedValue — YDisplay

Short string representing the current state of the function.

dnp string **AdvertisedValue**

display→Brightness YDisplay

Luminosity of the module informative LEDs (from 0 to 100).

dnp int **Brightness**

Writable. Changes the brightness of the display. The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

display→DisplayHeight — YDisplay

Display height, in pixels.

dnp int **DisplayHeight**

display→DisplayType — YDisplay

Display type: monochrome, gray levels or full color.

dnp int **DisplayType**

display→DisplayWidth — YDisplay

Display width, in pixels.

dnp int **DisplayWidth**

display→FriendlyName — YDisplay

Global identifier of the function in the format MODULE_NAME.FUNCTION_NAME.

dnp string **FriendlyName**

The returned string uses the logical names of the module and of the function if they are defined, otherwise the serial number of the module and the hardware identifier of the function (for example: MyCustomName.relay1)

display→FunctionId YDisplay

Hardware identifier of the display, without reference to the module.

dnp string **FunctionId**

For example `relay1`

display→HardwareId — YDisplay

Unique hardware identifier of the function in the form SERIAL.FUNCTIONID.

dnp string **HardwareId**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example RELAYLO1-123456.relay1).

display→IsOnline YDisplay

Checks if the function is currently reachable.

dnp bool **IsOnline**

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the function.

display→LayerCount — YDisplay

Number of available layers to draw on.

dnp int **LayerCount**

display→LayerHeight YDisplay

Height of the layers to draw on, in pixels.

dnp int **LayerHeight**

display→LayerWidth — YDisplay

Width of the layers to draw on, in pixels.

dnp int **LayerWidth**

display→LogicalName YDisplay

Logical name of the function.

dnp string **LogicalName**

Writable. You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

display→Orientation — YDisplay

Currently selected display orientation.

dnp int **Orientation**

Writable. Changes the display orientation. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

display→SerialNumber — YDisplay

Serial number of the module, as set by the factory.

dnp string **SerialNumber**

display→StartupSeq YDisplay

Name of the sequence to play when the displayed is powered on.

dnp string **StartupSeq**

Writable. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

display→clearCache() YDisplay

Invalidates the cache.

js	function **clearCache**( )
cpp	void **clearCache**( )
m	-(void) **clearCache**
pas	**clearCache**( )
vb	procedure **clearCache**( )
cs	void **clearCache**( )
java	void **clearCache**( )
py	**clearCache**( )
php	function **clearCache**( )
ts	async **clearCache**( ): Promise<void>
es	async **clearCache**( )

Invalidates the cache of the display attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

display→copyLayerContent() YDisplay

Copies the whole content of a layer to another layer.

```
js    function copyLayerContent( srcLayerId, dstLayerId)
cpp   int copyLayerContent( int srcLayerId, int dstLayerId)
m     -(int) copyLayerContent : (int) srcLayerId
                              : (int) dstLayerId
pas   LongInt copyLayerContent( srcLayerId: LongInt,
                                dstLayerId: LongInt): LongInt
vb    function copyLayerContent( ByVal srcLayerId As Integer,
                                 ByVal dstLayerId As Integer) As Integer
cs    int copyLayerContent( int srcLayerId, int dstLayerId)
java  int copyLayerContent( int srcLayerId, int dstLayerId)
uwp   async Task<int> copyLayerContent( int srcLayerId, int dstLayerId)
py    copyLayerContent( srcLayerId, dstLayerId)
php   function copyLayerContent( $srcLayerId, $dstLayerId)
ts    async copyLayerContent( srcLayerId: number, dstLayerId: number): Promise<number>
es    async copyLayerContent( srcLayerId, dstLayerId)
dnp   int copyLayerContent( int srcLayerId, int dstLayerId)
cp    int copyLayerContent( int srcLayerId, int dstLayerId)
cmd   YDisplay target copyLayerContent srcLayerId dstLayerId
```

The color and transparency of all the pixels from the destination layer are set to match the source pixels. This method only affects the displayed content, but does not change any property of the layer object. Note that layer 0 has no transparency support (it is always completely opaque).

Parameters :

srcLayerId the identifier of the source layer (a number in range 0..layerCount-1)

dstLayerId the identifier of the destination layer (a number in range 0..layerCount-1)

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→describe() YDisplay

Returns a short text that describes unambiguously the instance of the display in the form TYPE(NAME)=SERIAL.FUNCTIONID.

```
js      function describe( )
cpp     string describe( )
m       -(NSString*) describe
pas     string describe( ): string
vb      function describe( ) As String
cs      string describe( )
java    String describe( )
py      describe( )
php     function describe( )
ts      async describe( ): Promise<string>
es      async describe( )
```

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :
a string that describes the display (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

display→fade() YDisplay

Smoothly changes the brightness of the screen to produce a fade-in or fade-out effect.

```
js      function fade( brightness, duration)
cpp     int fade( int brightness, int duration)
m       -(int) fade : (int) brightness : (int) duration
pas     LongInt fade( brightness: LongInt, duration: LongInt): LongInt
vb      function fade( ByVal brightness As Integer, ByVal duration As Integer) As Integer
cs      int fade( int brightness, int duration)
java    int fade( int brightness, int duration)
uwp     async Task<int> fade( int brightness, int duration)
py      fade( brightness, duration)
php     function fade( $brightness, $duration)
ts      async fade( brightness: number, duration: number): Promise<number>
es      async fade( brightness, duration)
dnp     int fade( int brightness, int duration)
cp      int fade( int brightness, int duration)
cmd     YDisplay target fade brightness duration
```

Parameters :

brightness the new screen brightness

duration duration of the brightness transition, in milliseconds.

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→get_advertisedValue()
display→advertisedValue()

YDisplay

Returns the current value of the display (no more than 6 characters).

js	function **get_advertisedValue**( )
cpp	string **get_advertisedValue**( )
m	-(NSString*) advertisedValue
pas	string **get_advertisedValue**( ): string
vb	function **get_advertisedValue**( ) As String
cs	string **get_advertisedValue**( )
java	String **get_advertisedValue**( )
uwp	async Task<string> **get_advertisedValue**( )
py	**get_advertisedValue**( )
php	function **get_advertisedValue**( )
ts	async **get_advertisedValue**( ): Promise<string>
es	async **get_advertisedValue**( )
dnp	string **get_advertisedValue**( )
cp	string **get_advertisedValue**( )
cmd	YDisplay **target get_advertisedValue**

Returns :

a string corresponding to the current value of the display (no more than 6 characters).

On failure, throws an exception or returns `YDisplay.ADVERTISEDVALUE_INVALID`.

display→get_brightness()
display→brightness()

YDisplay

Returns the luminosity of the module informative LEDs (from 0 to 100).

```
js      function get_brightness( )
cpp     int get_brightness( )
m       -(int) brightness
pas     LongInt get_brightness( ): LongInt
vb      function get_brightness( ) As Integer
cs      int get_brightness( )
java    int get_brightness( )
uwp     async Task<int> get_brightness( )
py      get_brightness( )
php     function get_brightness( )
ts      async get_brightness( ): Promise<number>
es      async get_brightness( )
dnp     int get_brightness( )
cp      int get_brightness( )
cmd     YDisplay target get_brightness
```

Returns :

an integer corresponding to the luminosity of the module informative LEDs (from 0 to 100)

On failure, throws an exception or returns `YDisplay.BRIGHTNESS_INVALID`.

display→get_displayHeight()
display→displayHeight()

YDisplay

Returns the display height, in pixels.

js	function **get_displayHeight**( )
cpp	int **get_displayHeight**( )
m	-(int) displayHeight
pas	LongInt **get_displayHeight**( ): LongInt
vb	function **get_displayHeight**( ) As Integer
cs	int **get_displayHeight**( )
java	int **get_displayHeight**( )
uwp	async Task<int> **get_displayHeight**( )
py	**get_displayHeight**( )
php	function **get_displayHeight**( )
ts	async **get_displayHeight**( ): Promise<number>
es	async **get_displayHeight**( )
dnp	int **get_displayHeight**( )
cp	int **get_displayHeight**( )
cmd	YDisplay **target get_displayHeight**

Returns :
an integer corresponding to the display height, in pixels

On failure, throws an exception or returns `YDisplay.DISPLAYHEIGHT_INVALID`.

display→get_displayLayer() YDisplay
display→displayLayer()

Returns a YDisplayLayer object that can be used to draw on the specified layer.

```
js      function get_displayLayer( layerId)
cpp     YDisplayLayer* get_displayLayer( int layerId)
m       -(YDisplayLayer*) displayLayer : (int) layerId
pas     TYDisplayLayer get_displayLayer( layerId: LongInt): TYDisplayLayer
vb      function get_displayLayer( ByVal layerId As Integer) As YDisplayLayer
cs      YDisplayLayer get_displayLayer( int layerId)
java    YDisplayLayer get_displayLayer( int layerId)
uwp     async Task<YDisplayLayer> get_displayLayer( int layerId)
py      get_displayLayer( layerId)
php     function get_displayLayer( $layerId)
ts      async get_displayLayer( layerId: number): Promise<YDisplayLayer>
es      async get_displayLayer( layerId)
dnp     YDisplayLayerProxy get_displayLayer( int layerId)
cp      YDisplayLayerProxy* get_displayLayer( int layerId)
```

The content is displayed only when the layer is active on the screen (and not masked by other overlapping layers).

Parameters :

layerId the identifier of the layer (a number in range 0..layerCount-1)

Returns :

an `YDisplayLayer` object

On failure, throws an exception or returns `null`.

display→get_displayType()
display→displayType()

YDisplay

Returns the display type: monochrome, gray levels or full color.

js	function **get_displayType**( )
cpp	Y_DISPLAYTYPE_enum **get_displayType**( )
m	-(Y_DISPLAYTYPE_enum) displayType
pas	Integer **get_displayType**( ): Integer
vb	function **get_displayType**( ) As Integer
cs	int **get_displayType**( )
java	int **get_displayType**( )
uwp	async Task<int> **get_displayType**( )
py	**get_displayType**( )
php	function **get_displayType**( )
ts	async **get_displayType**( ): Promise<YDisplay_DisplayType>
es	async **get_displayType**( )
dnp	int **get_displayType**( )
cp	int **get_displayType**( )
cmd	YDisplay **target get_displayType**

Returns :

a value among `YDisplay.DISPLAYTYPE_MONO`, `YDisplay.DISPLAYTYPE_GRAY` and `YDisplay.DISPLAYTYPE_RGB` corresponding to the display type: monochrome, gray levels or full color

On failure, throws an exception or returns `YDisplay.DISPLAYTYPE_INVALID`.

display→get_displayWidth()
display→displayWidth()

YDisplay

Returns the display width, in pixels.

```
js      function get_displayWidth( )
cpp     int get_displayWidth( )
m       -(int) displayWidth
pas     LongInt get_displayWidth( ): LongInt
vb      function get_displayWidth( ) As Integer
cs      int get_displayWidth( )
java    int get_displayWidth( )
uwp     async Task<int> get_displayWidth( )
py      get_displayWidth( )
php     function get_displayWidth( )
ts      async get_displayWidth( ): Promise<number>
es      async get_displayWidth( )
dnp     int get_displayWidth( )
cp      int get_displayWidth( )
cmd     YDisplay target get_displayWidth
```

Returns :
an integer corresponding to the display width, in pixels

On failure, throws an exception or returns `YDisplay.DISPLAYWIDTH_INVALID`.

display→get_enabled()
display→enabled()

YDisplay

Returns true if the screen is powered, false otherwise.

```
js      function get_enabled( )
cpp     Y_ENABLED_enum get_enabled( )
m       -(Y_ENABLED_enum) enabled
pas     Integer get_enabled( ): Integer
vb      function get_enabled( ) As Integer
cs      int get_enabled( )
java    int get_enabled( )
uwp     async Task<int> get_enabled( )
py      get_enabled( )
php     function get_enabled( )
ts      async get_enabled( ): Promise<YDisplay_Enabled>
es      async get_enabled( )
dnp     int get_enabled( )
cp      int get_enabled( )
cmd     YDisplay target get_enabled
```

Returns :

either `YDisplay.ENABLED_FALSE` or `YDisplay.ENABLED_TRUE`, according to true if the screen is powered, false otherwise

On failure, throws an exception or returns `YDisplay.ENABLED_INVALID`.

display→get_errorMessage()
display→errorMessage()

YDisplay

Returns the error message of the latest error with the display.

```
js      function get_errorMessage( )
cpp     string get_errorMessage( )
m       -(NSString*) errorMessage
pas     string get_errorMessage( ): string
vb      function get_errorMessage( ) As String
cs      string get_errorMessage( )
java    String get_errorMessage( )
py      get_errorMessage( )
php     function get_errorMessage( )
ts      get_errorMessage( ): string
es      get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the display object

display→get_errorType()
display→errorType()

YDisplay

Returns the numerical error code of the latest error with the display.

js	function **get_errorType**( )
cpp	YRETCODE **get_errorType**( )
m	-(YRETCODE) errorType
pas	YRETCODE **get_errorType**( ): YRETCODE
vb	function **get_errorType**( ) As YRETCODE
cs	YRETCODE **get_errorType**( )
java	int **get_errorType**( )
py	**get_errorType**( )
php	function **get_errorType**( )
ts	**get_errorType**( ): number
es	**get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the display object

display→get_friendlyName()
display→friendlyName()

YDisplay

Returns a global identifier of the display in the format `MODULE_NAME.FUNCTION_NAME`.

js	function **get_friendlyName**( )
cpp	string **get_friendlyName**( )
m	-(NSString*) friendlyName
cs	string **get_friendlyName**( )
java	String **get_friendlyName**( )
py	**get_friendlyName**( )
php	function **get_friendlyName**( )
ts	async **get_friendlyName**( ): Promise<string>
es	async **get_friendlyName**( )
dnp	string **get_friendlyName**( )
cp	string **get_friendlyName**( )

The returned string uses the logical names of the module and of the display if they are defined, otherwise the serial number of the module and the hardware identifier of the display (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the display using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `YDisplay.FRIENDLYNAME_INVALID`.

display→get_functionDescriptor() YDisplay
display→functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

js	function **get_functionDescriptor**( )
cpp	YFUN_DESCR **get_functionDescriptor**( )
m	-(YFUN_DESCR) functionDescriptor
pas	YFUN_DESCR **get_functionDescriptor**( ): YFUN_DESCR
vb	function **get_functionDescriptor**( ) As YFUN_DESCR
cs	YFUN_DESCR **get_functionDescriptor**( )
java	String **get_functionDescriptor**( )
py	**get_functionDescriptor**( )
php	function **get_functionDescriptor**( )
ts	async **get_functionDescriptor**( ): Promise<string>
es	async **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :
an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y$CLASSNAME$.FUNCTIONDESCRIPTOR_INVALID.

display→get_functionId()
display→functionId()

YDisplay

Returns the hardware identifier of the display, without reference to the module.

```
js      function get_functionId( )
cpp     string get_functionId( )
m       -(NSString*) functionId
vb      function get_functionId( ) As String
cs      string get_functionId( )
java    String get_functionId( )
py      get_functionId( )
php     function get_functionId( )
ts      async get_functionId( ): Promise<string>
es      async get_functionId( )
dnp     string get_functionId( )
cp      string get_functionId( )
```

For example `relay1`

Returns :
a string that identifies the display (ex: `relay1`)

On failure, throws an exception or returns `YDisplay.FUNCTIONID_INVALID`.

display→get_hardwareId() YDisplay
display→hardwareId()

Returns the unique hardware identifier of the display in the form SERIAL.FUNCTIONID.

js	function **get_hardwareId**( )
cpp	string **get_hardwareId**( )
m	-(NSString*) hardwareId
vb	function **get_hardwareId**( ) As String
cs	string **get_hardwareId**( )
java	String **get_hardwareId**( )
py	**get_hardwareId**( )
php	function **get_hardwareId**( )
ts	async **get_hardwareId**( ): Promise<string>
es	async **get_hardwareId**( )
dnp	string **get_hardwareId**( )
cp	string **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the display (for example RELAYLO1-123456.relay1).

Returns :
a string that uniquely identifies the display (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns YDisplay.HARDWAREID_INVALID.

display→get_layerCount()
display→layerCount()

YDisplay

Returns the number of available layers to draw on.

```
js      function get_layerCount( )
cpp     int get_layerCount( )
m       -(int) layerCount
pas     LongInt get_layerCount( ): LongInt
vb      function get_layerCount( ) As Integer
cs      int get_layerCount( )
java    int get_layerCount( )
uwp     async Task<int> get_layerCount( )
py      get_layerCount( )
php     function get_layerCount( )
ts      async get_layerCount( ): Promise<number>
es      async get_layerCount( )
dnp     int get_layerCount( )
cp      int get_layerCount( )
cmd     YDisplay target get_layerCount
```

Returns :

an integer corresponding to the number of available layers to draw on

On failure, throws an exception or returns `YDisplay.LAYERCOUNT_INVALID`.

display→get_layerHeight() YDisplay
display→layerHeight()

Returns the height of the layers to draw on, in pixels.

js	function **get_layerHeight**( )
cpp	int **get_layerHeight**( )
m	-(int) layerHeight
pas	LongInt **get_layerHeight**( ): LongInt
vb	function **get_layerHeight**( ) As Integer
cs	int **get_layerHeight**( )
java	int **get_layerHeight**( )
uwp	async Task<int> **get_layerHeight**( )
py	**get_layerHeight**( )
php	function **get_layerHeight**( )
ts	async **get_layerHeight**( ): Promise<number>
es	async **get_layerHeight**( )
dnp	int **get_layerHeight**( )
cp	int **get_layerHeight**( )
cmd	YDisplay **target get_layerHeight**

Returns :
an integer corresponding to the height of the layers to draw on, in pixels

On failure, throws an exception or returns `YDisplay.LAYERHEIGHT_INVALID`.

display→get_layerWidth()
display→layerWidth()

YDisplay

Returns the width of the layers to draw on, in pixels.

```
js      function get_layerWidth( )
cpp     int get_layerWidth( )
m       -(int) layerWidth
pas     LongInt get_layerWidth( ): LongInt
vb      function get_layerWidth( ) As Integer
cs      int get_layerWidth( )
java    int get_layerWidth( )
uwp     async Task<int> get_layerWidth( )
py      get_layerWidth( )
php     function get_layerWidth( )
ts      async get_layerWidth( ): Promise<number>
es      async get_layerWidth( )
dnp     int get_layerWidth( )
cp      int get_layerWidth( )
cmd     YDisplay target get_layerWidth
```

Returns :

an integer corresponding to the width of the layers to draw on, in pixels

On failure, throws an exception or returns `YDisplay.LAYERWIDTH_INVALID`.

display→get_logicalName()
display→logicalName()

YDisplay

Returns the logical name of the display.

js	function **get_logicalName**( )
cpp	string **get_logicalName**( )
m	-(NSString*) logicalName
pas	string **get_logicalName**( ): string
vb	function **get_logicalName**( ) As String
cs	string **get_logicalName**( )
java	String **get_logicalName**( )
uwp	async Task<string> **get_logicalName**( )
py	**get_logicalName**( )
php	function **get_logicalName**( )
ts	async **get_logicalName**( ): Promise<string>
es	async **get_logicalName**( )
dnp	string **get_logicalName**( )
cp	string **get_logicalName**( )
cmd	YDisplay **target get_logicalName**

Returns :
a string corresponding to the logical name of the display.

On failure, throws an exception or returns `YDisplay.LOGICALNAME_INVALID`.

display→get_module()
display→module()

YDisplay

Gets the YModule object for the device on which the function is located.

js	function **get_module**( )
cpp	YModule * **get_module**( )
m	-(YModule*) module
pas	TYModule **get_module**( ): TYModule
vb	function **get_module**( ) As YModule
cs	YModule **get_module**( )
java	YModule **get_module**( )
py	**get_module**( )
php	function **get_module**( )
ts	async **get_module**( ): Promise<YModule>
es	async **get_module**( )
dnp	YModuleProxy **get_module**( )
cp	YModuleProxy * **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

display→get_module_async() YDisplay
display→module_async()

Gets the YModule object for the device on which the function is located (asynchronous version).

```
js  function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as on-line.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking Firefox JavaScript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous JavasSript calls for more details.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

display→get_orientation()
display→orientation()

YDisplay

Returns the currently selected display orientation.

```
js      function get_orientation( )
cpp     Y_ORIENTATION_enum get_orientation( )
m       -(Y_ORIENTATION_enum) orientation
pas     Integer get_orientation( ): Integer
vb      function get_orientation( ) As Integer
cs      int get_orientation( )
java    int get_orientation( )
uwp     async Task<int> get_orientation( )
py      get_orientation( )
php     function get_orientation( )
ts      async get_orientation( ): Promise<YDisplay_Orientation>
es      async get_orientation( )
dnp     int get_orientation( )
cp      int get_orientation( )
cmd     YDisplay target get_orientation
```

Returns :

a value among `YDisplay.ORIENTATION_LEFT`, `YDisplay.ORIENTATION_UP`, `YDisplay.ORIENTATION_RIGHT` and `YDisplay.ORIENTATION_DOWN` corresponding to the currently selected display orientation

On failure, throws an exception or returns `YDisplay.ORIENTATION_INVALID`.

display→get_serialNumber() YDisplay
display→serialNumber()

Returns the serial number of the module, as set by the factory.

```
js      function get_serialNumber( )
cpp     string get_serialNumber( )
m       -(NSString*) serialNumber
pas     string get_serialNumber( ): string
vb      function get_serialNumber( ) As String
cs      string get_serialNumber( )
java    String get_serialNumber( )
uwp     async Task<string> get_serialNumber( )
py      get_serialNumber( )
php     function get_serialNumber( )
ts      async get_serialNumber( ): Promise<string>
es      async get_serialNumber( )
dnp     string get_serialNumber( )
cp      string get_serialNumber( )
cmd     YDisplay target get_serialNumber
```

Returns :

a string corresponding to the serial number of the module, as set by the factory.

On failure, throws an exception or returns YFunction.SERIALNUMBER_INVALID.

display→get_startupSeq()
display→startupSeq()

YDisplay

Returns the name of the sequence to play when the displayed is powered on.

```
js      function get_startupSeq( )
cpp     string get_startupSeq( )
m       -(NSString*) startupSeq
pas     string get_startupSeq( ): string
vb      function get_startupSeq( ) As String
cs      string get_startupSeq( )
java    String get_startupSeq( )
uwp     async Task<string> get_startupSeq( )
py      get_startupSeq( )
php     function get_startupSeq( )
ts      async get_startupSeq( ): Promise<string>
es      async get_startupSeq( )
dnp     string get_startupSeq( )
cp      string get_startupSeq( )
cmd     YDisplay target get_startupSeq
```

Returns :

a string corresponding to the name of the sequence to play when the displayed is powered on

On failure, throws an exception or returns `YDisplay.STARTUPSEQ_INVALID`.

display→get_userData()
display→userData()

YDisplay

Returns the value of the userData attribute, as previously stored using method `set_userData`.

js	function **get_userData**( )
cpp	void * **get_userData**( )
m	-(id) userData
pas	Tobject **get_userData**( ): Tobject
vb	function **get_userData**( ) As Object
cs	object **get_userData**( )
java	Object **get_userData**( )
py	**get_userData**( )
php	function **get_userData**( )
ts	async **get_userData**( ): Promise<object|null>
es	async **get_userData**( )

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

display→isOnline() YDisplay

Checks if the display is currently reachable, without raising any error.

```
js      function isOnline( )
cpp     bool isOnline( )
m       -(BOOL) isOnline
pas     boolean isOnline( ): boolean
vb      function isOnline( ) As Boolean
cs      bool isOnline( )
java    boolean isOnline( )
py      isOnline( )
php     function isOnline( )
ts      async isOnline( ): Promise<boolean>
es      async isOnline( )
dnp     bool isOnline( )
cp      bool isOnline( )
```

If there is a cached value for the display in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the display.

Returns :
`true` if the display can be reached, and `false` otherwise

display→isOnline_async() YDisplay

Checks if the display is currently reachable, without raising any error (asynchronous version).

```
js  function isOnline_async( callback, context)
```

If there is a cached value for the display in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

display→isReadOnly() YDisplay

Test if the function is readOnly.

cpp	bool **isReadOnly**( )
m	-(bool) **isReadOnly**
pas	boolean **isReadOnly**( ): boolean
vb	function **isReadOnly**( ) As Boolean
cs	bool **isReadOnly**( )
java	boolean **isReadOnly**( )
uwp	async Task<bool> **isReadOnly**( )
py	**isReadOnly**( )
php	function **isReadOnly**( )
ts	async **isReadOnly**( ): Promise<boolean>
es	async **isReadOnly**( )
dnp	bool **isReadOnly**( )
cp	bool **isReadOnly**( )
cmd	YDisplay **target isReadOnly**

Return `true` if the function is write protected or that the function is not available.

Returns :
`true` if the function is readOnly or not online.

display→load() YDisplay

Preloads the display cache with a specified validity duration.

```
js      function load( msValidity)
cpp     YRETCODE load( int msValidity)
m       -(YRETCODE) load : (u64) msValidity
pas     YRETCODE load( msValidity: u64): YRETCODE
vb      function load( ByVal msValidity As Long) As YRETCODE
cs      YRETCODE load( ulong msValidity)
java    int load( long msValidity)
py      load( msValidity)
php     function load( $msValidity)
ts      async load( msValidity: number): Promise<number>
es      async load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI.SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

display→loadAttribute() YDisplay

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
js      function loadAttribute( attrName)
cpp     string loadAttribute( string attrName)
m       -(NSString*) loadAttribute : (NSString*) attrName
pas     string loadAttribute( attrName: string): string
vb      function loadAttribute( ByVal attrName As String) As String
cs      string loadAttribute( string attrName)
java    String loadAttribute( String attrName)
uwp     async Task<string> loadAttribute( string attrName)
py      loadAttribute( attrName)
php     function loadAttribute( $attrName)
ts      async loadAttribute( attrName: string): Promise<string>
es      async loadAttribute( attrName)
dnp     string loadAttribute( string attrName)
cp      string loadAttribute( string attrName)
```

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

display→load_async() YDisplay

Preloads the display cache with a specified validity duration (asynchronous version).

```
js  function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking the JavaScript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or YAPI.SUCCESS)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

display→muteValueCallbacks() YDisplay

Disables the propagation of every new advertised value to the parent hub.

```
js      function muteValueCallbacks( )
cpp     int muteValueCallbacks( )
m       -(int) muteValueCallbacks
pas     LongInt muteValueCallbacks( ): LongInt
vb      function muteValueCallbacks( ) As Integer
cs      int muteValueCallbacks( )
java    int muteValueCallbacks( )
uwp     async Task<int> muteValueCallbacks( )
py      muteValueCallbacks( )
php     function muteValueCallbacks( )
ts      async muteValueCallbacks( ): Promise<number>
es      async muteValueCallbacks( )
dnp     int muteValueCallbacks( )
cp      int muteValueCallbacks( )
cmd     YDisplay target muteValueCallbacks
```

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI.SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

display→newSequence() YDisplay

Starts to record all display commands into a sequence, for later replay.

js	function **newSequence**( )
cpp	int **newSequence**( )
m	-(int) **newSequence**
pas	LongInt **newSequence**( ): LongInt
vb	function **newSequence**( ) As Integer
cs	int **newSequence**( )
java	int **newSequence**( )
uwp	async Task<int> **newSequence**( )
py	**newSequence**( )
php	function **newSequence**( )
ts	async **newSequence**( ): Promise<number>
es	async **newSequence**( )
dnp	int **newSequence**( )
cp	int **newSequence**( )
cmd	YDisplay **target newSequence**

The name used to store the sequence is specified when calling `saveSequence()`, once the recording is complete.

Returns :
`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→nextDisplay() YDisplay

Continues the enumeration of displays started using `yFirstDisplay()`.

js	function **nextDisplay**( )
cpp	YDisplay * **nextDisplay**( )
m	-(nullable YDisplay*) **nextDisplay**
pas	TYDisplay **nextDisplay**( ): TYDisplay
vb	function **nextDisplay**( ) As YDisplay
cs	YDisplay **nextDisplay**( )
java	YDisplay **nextDisplay**( )
uwp	YDisplay **nextDisplay**( )
py	**nextDisplay**( )
php	function **nextDisplay**( )
ts	**nextDisplay**( ): YDisplay | null
es	**nextDisplay**( )

Caution: You can't make any assumption about the returned displays order. If you want to find a specific a display, use `Display.findDisplay()` and a hardwareID or a logical name.

Returns :

a pointer to a `YDisplay` object, corresponding to a display currently online, or a `null` pointer if there are no more displays to enumerate.

display→pauseSequence() YDisplay

Waits for a specified delay (in milliseconds) before playing next commands in current sequence.

```
js    function pauseSequence( delay_ms)
cpp   int pauseSequence( int delay_ms)
m     -(int) pauseSequence : (int) delay_ms
pas   LongInt pauseSequence( delay_ms: LongInt): LongInt
vb    function pauseSequence( ByVal delay_ms As Integer) As Integer
cs    int pauseSequence( int delay_ms)
java  int pauseSequence( int delay_ms)
uwp   async Task<int> pauseSequence( int delay_ms)
py    pauseSequence( delay_ms)
php   function pauseSequence( $delay_ms)
ts    async pauseSequence( delay_ms: number): Promise<number>
es    async pauseSequence( delay_ms)
dnp   int pauseSequence( int delay_ms)
cp    int pauseSequence( int delay_ms)
cmd   YDisplay target pauseSequence delay_ms
```

This method can be used while recording a display sequence, to insert a timed wait in the sequence (without any immediate effect). It can also be used dynamically while playing a pre-recorded sequence, to suspend or resume the execution of the sequence. To cancel a delay, call the same method with a zero delay.

Parameters :

delay_ms the duration to wait, in milliseconds

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→playSequence() YDisplay

Replays a display sequence previously recorded using `newSequence()` and `saveSequence()`.

```
js      function playSequence( sequenceName)
cpp     int playSequence( string sequenceName)
m       -(int) playSequence : (NSString*) sequenceName
pas     LongInt playSequence( sequenceName: string): LongInt
vb      function playSequence( ByVal sequenceName As String) As Integer
cs      int playSequence( string sequenceName)
java    int playSequence( String sequenceName)
uwp     async Task<int> playSequence( string sequenceName)
py      playSequence( sequenceName)
php     function playSequence( $sequenceName)
ts      async playSequence( sequenceName: string): Promise<number>
es      async playSequence( sequenceName)
dnp     int playSequence( string sequenceName)
cp      int playSequence( string sequenceName)
cmd     YDisplay target playSequence sequenceName
```

Parameters :

sequenceName the name of the newly created sequence

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→registerValueCallback() YDisplay

Registers the callback function that is invoked on every change of advertised value.

js	function **registerValueCallback**(**callback**)
cpp	int **registerValueCallback**(YDisplayValueCallback **callback**)
m	-(int) **registerValueCallback** : (YDisplayValueCallback _Nullable) **callback**
pas	LongInt **registerValueCallback**(**callback**: TYDisplayValueCallback): LongInt
vb	function **registerValueCallback**(ByVal **callback** As YDisplayValueCallback) As Integer
cs	int **registerValueCallback**(ValueCallback **callback**)
java	int **registerValueCallback**(UpdateCallback **callback**)
uwp	async Task<int> **registerValueCallback**(ValueCallback **callback**)
py	**registerValueCallback**(**callback**)
php	function **registerValueCallback**(**$callback**)
ts	async **registerValueCallback**(**callback**: YDisplayValueCallback | null): Promise<number>
es	async **registerValueCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

display→resetAll() YDisplay

Clears the display screen and resets all display layers to their default state.

```
js      function resetAll( )
cpp     int resetAll( )
m       -(int) resetAll
pas     LongInt resetAll( ): LongInt
vb      function resetAll( ) As Integer
cs      int resetAll( )
java    int resetAll( )
uwp     async Task<int> resetAll( )
py      resetAll( )
php     function resetAll( )
ts      async resetAll( ): Promise<number>
es      async resetAll( )
dnp     int resetAll( )
cp      int resetAll( )
cmd     YDisplay target resetAll
```

Using this function in a sequence will kill the sequence play-back. Don't use that function to reset the display at sequence start-up.

Returns :
YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→saveSequence() YDisplay

Stops recording display commands and saves the sequence into the specified file on the display internal memory.

```
js      function saveSequence( sequenceName)
cpp     int saveSequence( string sequenceName)
m       -(int) saveSequence : (NSString*) sequenceName
pas     LongInt saveSequence( sequenceName: string): LongInt
vb      function saveSequence( ByVal sequenceName As String) As Integer
cs      int saveSequence( string sequenceName)
java    int saveSequence( String sequenceName)
uwp     async Task<int> saveSequence( string sequenceName)
py      saveSequence( sequenceName)
php     function saveSequence( $sequenceName)
ts      async saveSequence( sequenceName: string): Promise<number>
es      async saveSequence( sequenceName)
dnp     int saveSequence( string sequenceName)
cp      int saveSequence( string sequenceName)
cmd     YDisplay target saveSequence sequenceName
```

The sequence can be later replayed using `playSequence()`.

Parameters :

sequenceName the name of the newly created sequence

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→set_brightness()
display→setBrightness()

YDisplay

Changes the brightness of the display.

```
js      function set_brightness( newval)
cpp     int set_brightness( int newval)
m       -(int) setBrightness : (int) newval
pas     integer set_brightness( newval: LongInt): integer
vb      function set_brightness( ByVal newval As Integer) As Integer
cs      int set_brightness( int newval)
java    int set_brightness( int newval)
uwp     async Task<int> set_brightness( int newval)
py      set_brightness( newval)
php     function set_brightness( $newval)
ts      async set_brightness( newval: number): Promise<number>
es      async set_brightness( newval)
dnp     int set_brightness( int newval)
cp      int set_brightness( int newval)
cmd     YDisplay target set_brightness newval
```

The parameter is a value between 0 and 100. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer corresponding to the brightness of the display

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→set_enabled() YDisplay
display→setEnabled()

Changes the power state of the display.

js	function **set_enabled**(**newval**)
cpp	int **set_enabled**(Y_ENABLED_enum **newval**)
m	-(int) setEnabled : (Y_ENABLED_enum) **newval**
pas	integer **set_enabled**(**newval**: Integer): integer
vb	function **set_enabled**(ByVal **newval** As Integer) As Integer
cs	int **set_enabled**(int **newval**)
java	int **set_enabled**(int **newval**)
uwp	async Task<int> **set_enabled**(int **newval**)
py	**set_enabled**(**newval**)
php	function **set_enabled**(**$newval**)
ts	async **set_enabled**(**newval**: YDisplay_Enabled): Promise<number>
es	async **set_enabled**(**newval**)
dnp	int **set_enabled**(int **newval**)
cp	int **set_enabled**(int **newval**)
cmd	YDisplay **target set_enabled newval**

Parameters :

newval either `YDisplay.ENABLED_FALSE` or `YDisplay.ENABLED_TRUE`, according to the power state of the display

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→set_logicalName()
display→setLogicalName()

YDisplay

Changes the logical name of the display.

js	function **set_logicalName**(**newval**)
cpp	int **set_logicalName**(string **newval**)
m	-(int) setLogicalName : (NSString*) **newval**
pas	integer **set_logicalName**(**newval**: string): integer
vb	function **set_logicalName**(ByVal **newval** As String) As Integer
cs	int **set_logicalName**(string **newval**)
java	int **set_logicalName**(String **newval**)
uwp	async Task<int> **set_logicalName**(string **newval**)
py	**set_logicalName**(**newval**)
php	function **set_logicalName**(**$newval**)
ts	async **set_logicalName**(**newval**: string): Promise<number>
es	async **set_logicalName**(**newval**)
dnp	int **set_logicalName**(string **newval**)
cp	int **set_logicalName**(string **newval**)
cmd	YDisplay **target set_logicalName newval**

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the display.

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→set_orientation() YDisplay
display→setOrientation()

Changes the display orientation.

js	function **set_orientation**(**newval**)
cpp	int **set_orientation**(Y_ORIENTATION_enum **newval**)
m	-(int) setOrientation : (Y_ORIENTATION_enum) **newval**
pas	integer **set_orientation**(**newval**: Integer): integer
vb	function **set_orientation**(ByVal **newval** As Integer) As Integer
cs	int **set_orientation**(int **newval**)
java	int **set_orientation**(int **newval**)
uwp	async Task<int> **set_orientation**(int **newval**)
py	**set_orientation**(**newval**)
php	function **set_orientation**(**$newval**)
ts	async **set_orientation**(**newval**: YDisplay_Orientation): Promise<number>
es	async **set_orientation**(**newval**)
dnp	int **set_orientation**(int **newval**)
cp	int **set_orientation**(int **newval**)
cmd	YDisplay **target set_orientation newval**

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a value among `YDisplay.ORIENTATION_LEFT`, `YDisplay.ORIENTATION_UP`, `YDisplay.ORIENTATION_RIGHT` and `YDisplay.ORIENTATION_DOWN` corresponding to the display orientation

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→set_startupSeq() YDisplay
display→setStartupSeq()

Changes the name of the sequence to play when the displayed is powered on.

```
js      function set_startupSeq( newval)
cpp     int set_startupSeq( string newval)
m       -(int) setStartupSeq : (NSString*) newval
pas     integer set_startupSeq( newval: string): integer
vb      function set_startupSeq( ByVal newval As String) As Integer
cs      int set_startupSeq( string newval)
java    int set_startupSeq( String newval)
uwp     async Task<int> set_startupSeq( string newval)
py      set_startupSeq( newval)
php     function set_startupSeq( $newval)
ts      async set_startupSeq( newval: string): Promise<number>
es      async set_startupSeq( newval)
dnp     int set_startupSeq( string newval)
cp      int set_startupSeq( string newval)
cmd     YDisplay target set_startupSeq newval
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the name of the sequence to play when the displayed is powered on

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→set_userData() YDisplay
display→setUserData()

Stores a user context provided as argument in the userData attribute of the function.

```
js      function set_userData( data)
cpp     void set_userData( void * data)
m       -(void) setUserData : (id) data
pas     set_userData( data: Tobject)
vb      procedure set_userData( ByVal data As Object)
cs      void set_userData( object data)
java    void set_userData( Object data)
py      set_userData( data)
php     function set_userData( $data)
ts      async set_userData( data: object|null): Promise<void>
es      async set_userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

display→stopSequence() YDisplay

Stops immediately any ongoing sequence replay.

```
js      function stopSequence( )
cpp     int stopSequence( )
m       -(int) stopSequence
pas     LongInt stopSequence( ): LongInt
vb      function stopSequence( ) As Integer
cs      int stopSequence( )
java    int stopSequence( )
uwp     async Task<int> stopSequence( )
py      stopSequence( )
php     function stopSequence( )
ts      async stopSequence( ): Promise<number>
es      async stopSequence( )
dnp     int stopSequence( )
cp      int stopSequence( )
cmd     YDisplay target stopSequence
```

The display is left as is.

Returns :
YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→swapLayerContent() YDisplay

Swaps the whole content of two layers.

```
js    function swapLayerContent( layerIdA, layerIdB)
cpp   int swapLayerContent( int layerIdA, int layerIdB)
m     -(int) swapLayerContent : (int) layerIdA
                              : (int) layerIdB
pas   LongInt swapLayerContent( layerIdA: LongInt, layerIdB: LongInt): LongInt
vb    function swapLayerContent( ByVal layerIdA As Integer,
                                 ByVal layerIdB As Integer) As Integer
cs    int swapLayerContent( int layerIdA, int layerIdB)
java  int swapLayerContent( int layerIdA, int layerIdB)
uwp   async Task<int> swapLayerContent( int layerIdA, int layerIdB)
py    swapLayerContent( layerIdA, layerIdB)
php   function swapLayerContent( $layerIdA, $layerIdB)
ts    async swapLayerContent( layerIdA: number, layerIdB: number): Promise<number>
es    async swapLayerContent( layerIdA, layerIdB)
dnp   int swapLayerContent( int layerIdA, int layerIdB)
cp    int swapLayerContent( int layerIdA, int layerIdB)
cmd   YDisplay target swapLayerContent layerIdA layerIdB
```

The color and transparency of all the pixels from the two layers are swapped. This method only affects the displayed content, but does not change any property of the layer objects. In particular, the visibility of each layer stays unchanged. When used between one hidden layer and a visible layer, this method makes it possible to easily implement double-buffering. Note that layer 0 has no transparency support (it is always completely opaque).

Parameters :

layerIdA the first layer (a number in range 0..layerCount-1)

layerIdB the second layer (a number in range 0..layerCount-1)

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→unmuteValueCallbacks() YDisplay

Re-enables the propagation of every new advertised value to the parent hub.

```
js      function unmuteValueCallbacks( )
cpp     int unmuteValueCallbacks( )
m       -(int) unmuteValueCallbacks
pas     LongInt unmuteValueCallbacks( ): LongInt
vb      function unmuteValueCallbacks( ) As Integer
cs      int unmuteValueCallbacks( )
java    int unmuteValueCallbacks( )
uwp     async Task<int> unmuteValueCallbacks( )
py      unmuteValueCallbacks( )
php     function unmuteValueCallbacks( )
ts      async unmuteValueCallbacks( ): Promise<number>
es      async unmuteValueCallbacks( )
dnp     int unmuteValueCallbacks( )
cp      int unmuteValueCallbacks( )
cmd     YDisplay target unmuteValueCallbacks
```

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI.SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

display→upload() YDisplay

Uploads an arbitrary file (for instance a GIF file) to the display, to the specified full path name.

```
js      function upload( pathname, content)
cpp     int upload( string pathname, string content)
m       -(int) upload : (NSString*) pathname
                      : (NSData*) content
pas     LongInt upload( pathname: string, content: TByteArray): LongInt
vb      procedure upload( ByVal pathname As String, ByVal content As Byte()
cs      int upload( string pathname, byte[] content)
java    int upload( String pathname, byte[] content)
uwp     async Task<int> upload( string pathname, byte[] content)
py      upload( pathname, content)
php     function upload( $pathname, $content)
ts      async upload( pathname: string, content: Uint8Array): Promise<number>
es      async upload( pathname, content)
dnp     int upload( string pathname, byte[] content)
cp      int upload( string pathname, string content)
cmd     YDisplay target upload pathname content
```

If a file already exists with the same path name, its content is overwritten.

Parameters :

pathname path and name of the new file to create

content binary buffer with the content to set

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

display→wait_async() YDisplay

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js  function wait_async( callback, context)
ts  wait_async( callback: Function, context: object)
es  wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

24.4. Class YDisplayLayer

Interface for drawing into display layers, obtained by calling `display.get_displayLayer`.

Each `DisplayLayer` represents an image layer containing objects to display (bitmaps, text, etc.). The content is displayed only when the layer is active on the screen (and not masked by other overlapping layers).

In order to use the functions described here, you should include:

`js`	<script type='text/javascript' src='yocto_display.js'></script>
`cpp`	#include "yocto_display.h"
`m`	#import "yocto_display.h"
`pas`	uses yocto_display;
`vb`	yocto_display.vb
`cs`	yocto_display.cs
`java`	import com.yoctopuce.YoctoAPI.YDisplayLayer;
`uwp`	import com.yoctopuce.YoctoAPI.YDisplayLayer;
`py`	from yocto_display import *
`php`	require_once('yocto_display.php');
`ts`	in HTML: import { YDisplayLayer } from '../../dist/esm/yocto_display.js'; in Node.js: import { YDisplayLayer } from 'yoctolib-cjs/yocto_display.js';
`es`	in HTML: <script src="../../lib/yocto_display.js"></script> in node.js: require('yoctolib-es2017/yocto_display.js');
`dnp`	import YoctoProxyAPI.YDisplayLayerProxy
`cp`	#include "yocto_display_proxy.h"
`ml`	import YoctoProxyAPI.YDisplayLayerProxy

`YDisplayLayer` methods

displaylayer→clear()

Erases the whole content of the layer (makes it fully transparent).

displaylayer→clearConsole()

Blanks the console area within console margins, and resets the console pointer to the upper left corner of the console.

displaylayer→consoleOut(**text**)

Outputs a message in the console area, and advances the console pointer accordingly.

displaylayer→drawBar(**x1**, **y1**, **x2**, **y2**)

Draws a filled rectangular bar at a specified position.

displaylayer→drawBitmap(**x**, **y**, **w**, **bitmap**, **bgcol**)

Draws a bitmap at the specified position.

displaylayer→drawCircle(**x**, **y**, **r**)

Draws an empty circle at a specified position.

displaylayer→drawDisc(**x**, **y**, **r**)

Draws a filled disc at a given position.

displaylayer→drawImage(**x**, **y**, **imagename**)

Draws a GIF image at the specified position.

displaylayer→drawPixel(**x**, **y**)

Draws a single pixel at the specified position.

displaylayer→drawRect(**x1**, **y1**, **x2**, **y2**)

Draws an empty rectangle at a specified position.

displaylayer→drawText(x, y, anchor, text)
Draws a text string at the specified position.

displaylayer→get_display()
Gets parent YDisplay.

displaylayer→get_displayHeight()
Returns the display height, in pixels.

displaylayer→get_displayWidth()
Returns the display width, in pixels.

displaylayer→get_layerHeight()
Returns the height of the layers to draw on, in pixels.

displaylayer→get_layerWidth()
Returns the width of the layers to draw on, in pixels.

displaylayer→hide()
Hides the layer.

displaylayer→lineTo(x, y)
Draws a line from current drawing pointer position to the specified position.

displaylayer→moveTo(x, y)
Moves the drawing pointer of this layer to the specified position.

displaylayer→reset()
Reverts the layer to its initial state (fully transparent, default settings).

displaylayer→selectColorPen(color)
Selects the pen color for all subsequent drawing functions, including text drawing.

displaylayer→selectEraser()
Selects an eraser instead of a pen for all subsequent drawing functions, except for bitmap copy functions.

displaylayer→selectFont(fontname)
Selects a font to use for the next text drawing functions, by providing the name of the font file.

displaylayer→selectGrayPen(graylevel)
Selects the pen gray level for all subsequent drawing functions, including text drawing.

displaylayer→setAntialiasingMode(mode)
Enables or disables anti-aliasing for drawing oblique lines and circles.

displaylayer→setConsoleBackground(bgcol)
Sets up the background color used by the `clearConsole` function and by the console scrolling feature.

displaylayer→setConsoleMargins(x1, y1, x2, y2)
Sets up display margins for the `consoleOut` function.

displaylayer→setConsoleWordWrap(wordwrap)
Sets up the wrapping behavior used by the `consoleOut` function.

displaylayer→setLayerPosition(x, y, scrollTime)
Sets the position of the layer relative to the display upper left corner.

displaylayer→unhide()
Shows the layer.

displaylayer→clear() YDisplayLayer

Erases the whole content of the layer (makes it fully transparent).

```
js      function clear( )
cpp     int clear( )
m       -(int) clear
pas     LongInt clear( ): LongInt
vb      function clear( ) As Integer
cs      int clear( )
java    int clear( )
uwp     async Task<int> clear( )
py      clear( )
php     function clear( )
ts      async clear( ): Promise<number>
es      async clear( )
dnp     int clear( )
cp      int clear( )
cmd     YDisplay target [-layer layerId] clear
```

This method does not change any other attribute of the layer. To reinitialize the layer attributes to defaults settings, use the method `reset()` instead.

Returns :
`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→clearConsole() YDisplayLayer

Blanks the console area within console margins, and resets the console pointer to the upper left corner of the console.

```
js      function clearConsole( )
cpp     int clearConsole( )
m       -(int) clearConsole
pas     LongInt clearConsole( ): LongInt
vb      function clearConsole( ) As Integer
cs      int clearConsole( )
java    int clearConsole( )
uwp     async Task<int> clearConsole( )
py      clearConsole( )
php     function clearConsole( )
ts      async clearConsole( ): Promise<number>
es      async clearConsole( )
dnp     int clearConsole( )
cp      int clearConsole( )
cmd     YDisplay target [-layer layerId] clearConsole
```

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→consoleOut() YDisplayLayer

Outputs a message in the console area, and advances the console pointer accordingly.

```
js    function consoleOut( text)
cpp   int consoleOut( string text)
m     -(int) consoleOut : (NSString*) text
pas   LongInt consoleOut( text: string): LongInt
vb    function consoleOut( ByVal text As String) As Integer
cs    int consoleOut( string text)
java  int consoleOut( String text)
uwp   async Task<int> consoleOut( string text)
py    consoleOut( text)
php   function consoleOut( $text)
ts    async consoleOut( text: string): Promise<number>
es    async consoleOut( text)
dnp   int consoleOut( string text)
cp    int consoleOut( string text)
cmd   YDisplay target [-layer layerId] consoleOut  text
```

The console pointer position is automatically moved to the beginning of the next line when a newline character is met, or when the right margin is hit. When the new text to display extends below the lower margin, the console area is automatically scrolled up.

Parameters :

text the message to display

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawBar() YDisplayLayer

Draws a filled rectangular bar at a specified position.

```
js    function drawBar( x1, y1, x2, y2)
cpp   int drawBar( int x1, int y1, int x2, int y2)
m     -(int) drawBar : (int) x1
                     : (int) y1
                     : (int) x2
                     : (int) y2
pas   LongInt drawBar( x1: LongInt,
                       y1: LongInt,
                       x2: LongInt,
                       y2: LongInt): LongInt
vb    function drawBar( ByVal x1 As Integer,
                        ByVal y1 As Integer,
                        ByVal x2 As Integer,
                        ByVal y2 As Integer) As Integer
cs    int drawBar( int x1, int y1, int x2, int y2)
java  int drawBar( int x1, int y1, int x2, int y2)
uwp   async Task<int> drawBar( int x1, int y1, int x2, int y2)
py    drawBar( x1, y1, x2, y2)
php   function drawBar( $x1, $y1, $x2, $y2)
ts    async drawBar( x1: number, y1: number, x2: number, y2: number): Promise<number>
es    async drawBar( x1, y1, x2, y2)
dnp   int drawBar( int x1, int y1, int x2, int y2)
cp    int drawBar( int x1, int y1, int x2, int y2)
cmd   YDisplay target [-layer layerId] drawBar  x1 y1 x2 y2
```

Parameters :

x1 the distance from left of layer to the left border of the rectangle, in pixels
y1 the distance from top of layer to the top border of the rectangle, in pixels
x2 the distance from left of layer to the right border of the rectangle, in pixels
y2 the distance from top of layer to the bottom border of the rectangle, in pixels

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawBitmap() YDisplayLayer

Draws a bitmap at the specified position.

```
js      function drawBitmap( x, y, w, bitmap, bgcol)
cpp     int drawBitmap( int x, int y, int w, string bitmap, int bgcol)
m       -(int) drawBitmap : (int) x
                          : (int) y
                          : (int) w
                          : (NSData*) bitmap
                          : (int) bgcol
pas     LongInt drawBitmap( x: LongInt,
                            y: LongInt,
                            w: LongInt,
                            bitmap: TByteArray,
                            bgcol: LongInt): LongInt
vb      procedure drawBitmap( ByVal x As Integer,
                              ByVal y As Integer,
                              ByVal w As Integer,
                              ByVal bitmap As Byte()
cs      int drawBitmap( int x, int y, int w, byte[] bitmap, int bgcol)
java    int drawBitmap( int x, int y, int w, byte[] bitmap, int bgcol)
uwp     async Task<int> drawBitmap( int x,
                                    int y,
                                    int w,
                                    byte[] bitmap,
                                    int bgcol)
py      drawBitmap( x, y, w, bitmap, bgcol)
php     function drawBitmap( $x, $y, $w, $bitmap, $bgcol)
ts      async drawBitmap( x: number, y: number, w: number, bitmap: Uint8Array, bgcol: number):
                          Promise<number>
es      async drawBitmap( x, y, w, bitmap, bgcol)
dnp     int drawBitmap( int x, int y, int w, byte[] bitmap, int bgcol)
cp      int drawBitmap( int x, int y, int w, string bitmap, int bgcol)
cmd     YDisplay target [-layer layerId] drawBitmap  x y w bitmap bgcol
```

The bitmap is provided as a binary object, where each pixel maps to a bit, from left to right and from top to bottom. The most significant bit of each byte maps to the leftmost pixel, and the least significant bit maps to the rightmost pixel. Bits set to 1 are drawn using the layer selected pen color. Bits set to 0 are drawn using the specified background gray level, unless -1 is specified, in which case they are not drawn at all (as if transparent).

Parameters :

x	the distance from left of layer to the left of the bitmap, in pixels
y	the distance from top of layer to the top of the bitmap, in pixels
w	the width of the bitmap, in pixels
bitmap	a binary object
bgcol	the background gray level to use for zero bits (0 = black, 255 = white), or -1 to leave the pixels unchanged

Returns :
YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawCircle() YDisplayLayer

Draws an empty circle at a specified position.

```
js    function drawCircle( x, y, r)
cpp   int drawCircle( int x, int y, int r)
m     -(int) drawCircle : (int) x
                        : (int) y
                        : (int) r
pas   LongInt drawCircle( x: LongInt, y: LongInt, r: LongInt): LongInt
vb    function drawCircle( ByVal x As Integer, ByVal y As Integer, ByVal r As Integer) As Integer
cs    int drawCircle( int x, int y, int r)
java  int drawCircle( int x, int y, int r)
uwp   async Task<int> drawCircle( int x, int y, int r)
py    drawCircle( x, y, r)
php   function drawCircle( $x, $y, $r)
ts    async drawCircle( x: number, y: number, r: number): Promise<number>
es    async drawCircle( x, y, r)
dnp   int drawCircle( int x, int y, int r)
cp    int drawCircle( int x, int y, int r)
cmd   YDisplay target [-layer layerId] drawCircle  x y r
```

Parameters :

- **x** the distance from left of layer to the center of the circle, in pixels
- **y** the distance from top of layer to the center of the circle, in pixels
- **r** the radius of the circle, in pixels

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawDisc() YDisplayLayer

Draws a filled disc at a given position.

```
js    function drawDisc( x, y, r)
cpp   int drawDisc( int x, int y, int r)
m     -(int) drawDisc : (int) x
                      : (int) y
                      : (int) r
pas   LongInt drawDisc( x: LongInt, y: LongInt, r: LongInt): LongInt
vb    function drawDisc( ByVal x As Integer, ByVal y As Integer, ByVal r As Integer) As Integer
cs    int drawDisc( int x, int y, int r)
java  int drawDisc( int x, int y, int r)
uwp   async Task<int> drawDisc( int x, int y, int r)
py    drawDisc( x, y, r)
php   function drawDisc( $x, $y, $r)
ts    async drawDisc( x: number, y: number, r: number): Promise<number>
es    async drawDisc( x, y, r)
dnp   int drawDisc( int x, int y, int r)
cp    int drawDisc( int x, int y, int r)
cmd   YDisplay target [-layer layerId] drawDisc  x y r
```

Parameters :

x the distance from left of layer to the center of the disc, in pixels
y the distance from top of layer to the center of the disc, in pixels
r the radius of the disc, in pixels

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawImage() YDisplayLayer

Draws a GIF image at the specified position.

```
js      function drawImage( x, y, imagename)
cpp     int drawImage( int x, int y, string imagename)
m       -(int) drawImage : (int) x
                         : (int) y
                         : (NSString*) imagename
pas     LongInt drawImage( x: LongInt, y: LongInt, imagename: string): LongInt
vb      function drawImage( ByVal x As Integer,
                            ByVal y As Integer,
                            ByVal imagename As String) As Integer
cs      int drawImage( int x, int y, string imagename)
java    int drawImage( int x, int y, String imagename)
uwp     async Task<int> drawImage( int x, int y, string imagename)
py      drawImage( x, y, imagename)
php     function drawImage( $x, $y, $imagename)
ts      async drawImage( x: number, y: number, imagename: string): Promise<number>
es      async drawImage( x, y, imagename)
dnp     int drawImage( int x, int y, string imagename)
cp      int drawImage( int x, int y, string imagename)
cmd     YDisplay target [-layer layerId] drawImage  x y imagename
```

The GIF image must have been previously uploaded to the device built-in memory. If you experience problems using an image file, check the device logs for any error message such as missing image file or bad image file format.

Parameters :

x	the distance from left of layer to the left of the image, in pixels
y	the distance from top of layer to the top of the image, in pixels
imagename	the GIF file name

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawPixel() YDisplayLayer

Draws a single pixel at the specified position.

```
js      function drawPixel( x, y)
cpp     int drawPixel( int x, int y)
m       -(int) drawPixel : (int) x
                         : (int) y
pas     LongInt drawPixel( x: LongInt, y: LongInt): LongInt
vb      function drawPixel( ByVal x As Integer, ByVal y As Integer) As Integer
cs      int drawPixel( int x, int y)
java    int drawPixel( int x, int y)
uwp     async Task<int> drawPixel( int x, int y)
py      drawPixel( x, y)
php     function drawPixel( $x, $y)
ts      async drawPixel( x: number, y: number): Promise<number>
es      async drawPixel( x, y)
dnp     int drawPixel( int x, int y)
cp      int drawPixel( int x, int y)
cmd     YDisplay target [-layer layerId] drawPixel  x y
```

Parameters :

x the distance from left of layer, in pixels

y the distance from top of layer, in pixels

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawRect() YDisplayLayer

Draws an empty rectangle at a specified position.

```
js    function drawRect( x1, y1, x2, y2)
cpp   int drawRect( int x1, int y1, int x2, int y2)
m     -(int) drawRect : (int) x1
                      : (int) y1
                      : (int) x2
                      : (int) y2
pas   LongInt drawRect( x1: LongInt,
                        y1: LongInt,
                        x2: LongInt,
                        y2: LongInt): LongInt
vb    function drawRect( ByVal x1 As Integer,
                         ByVal y1 As Integer,
                         ByVal x2 As Integer,
                         ByVal y2 As Integer) As Integer
cs    int drawRect( int x1, int y1, int x2, int y2)
java  int drawRect( int x1, int y1, int x2, int y2)
uwp   async Task<int> drawRect( int x1, int y1, int x2, int y2)
py    drawRect( x1, y1, x2, y2)
php   function drawRect( $x1, $y1, $x2, $y2)
ts    async drawRect( x1: number, y1: number, x2: number, y2: number): Promise<number>
es    async drawRect( x1, y1, x2, y2)
dnp   int drawRect( int x1, int y1, int x2, int y2)
cp    int drawRect( int x1, int y1, int x2, int y2)
cmd   YDisplay target [-layer layerId] drawRect  x1 y1 x2 y2
```

Parameters :

x1 the distance from left of layer to the left border of the rectangle, in pixels
y1 the distance from top of layer to the top border of the rectangle, in pixels
x2 the distance from left of layer to the right border of the rectangle, in pixels
y2 the distance from top of layer to the bottom border of the rectangle, in pixels

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→drawText() YDisplayLayer

Draws a text string at the specified position.

```
js    function drawText( x, y, anchor, text)
cpp   int drawText( int x, int y, Y_ALIGN anchor, string text)
m     -(int) drawText : (int) x
                      : (int) y
                      : (Y_ALIGN) anchor
                      : (NSString*) text
pas   LongInt drawText( x: LongInt,
                        y: LongInt,
                        anchor: TYALIGN,
                        text: string): LongInt
vb    function drawText( ByVal x As Integer,
                         ByVal y As Integer,
                         ByVal anchor As Y_ALIGN,
                         ByVal text As String) As Integer
cs    int drawText( int x, int y, ALIGN anchor, string text)
java  int drawText( int x, int y, ALIGN anchor, String text)
uwp   async Task<int> drawText( int x, int y, ALIGN anchor, string text)
py    drawText( x, y, anchor, text)
php   function drawText( $x, $y, $anchor, $text)
ts    async drawText( x: number, y: number, anchor: YDisplayLayer_Align, text: string): Promise<number>
es    async drawText( x, y, anchor, text)
dnp   int drawText( int x, int y, int anchor, string text)
cp    int drawText( int x, int y, int anchor, string text)
cmd   YDisplay target [-layer layerId] drawText  x y anchor text
```

The point of the text that is aligned to the specified pixel position is called the anchor point, and can be chosen among several options. Text is rendered from left to right, without implicit wrapping.

Parameters :

x	the distance from left of layer to the text anchor point, in pixels
y	the distance from top of layer to the text anchor point, in pixels
anchor	the text anchor point, chosen among the `YDisplayLayer.ALIGN` enumeration: `YDisplayLayer.ALIGN_TOP_LEFT`, `YDisplayLayer.ALIGN_CENTER_LEFT`, `YDisplayLayer.ALIGN_BASELINE_LEFT`, `YDisplayLayer.ALIGN_BOTTOM_LEFT`, `YDisplayLayer.ALIGN_TOP_CENTER`, `YDisplayLayer.ALIGN_CENTER`, `YDisplayLayer.ALIGN_BASELINE_CENTER`, `YDisplayLayer.ALIGN_BOTTOM_CENTER`, `YDisplayLayer.ALIGN_TOP_DECIMAL`, `YDisplayLayer.ALIGN_CENTER_DECIMAL`, `YDisplayLayer.ALIGN_BASELINE_DECIMAL`, `YDisplayLayer.ALIGN_BOTTOM_DECIMAL`, `YDisplayLayer.ALIGN_TOP_RIGHT`, `YDisplayLayer.ALIGN_CENTER_RIGHT`, `YDisplayLayer.ALIGN_BASELINE_RIGHT`, `YDisplayLayer.ALIGN_BOTTOM_RIGHT`.
text	the text string to draw

Returns :
YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→get_display()
displaylayer→display()

YDisplayLayer

Gets parent YDisplay.

```
js      function get_display( )
cpp     YDisplay* get_display( )
m       -(YDisplay*) display
pas     TYDisplay get_display( ): TYDisplay
vb      function get_display( ) As YDisplay
cs      YDisplay get_display( )
java    YDisplay get_display( )
uwp     async Task<YDisplay> get_display( )
py      get_display( )
php     function get_display( )
ts      async get_display( ): Promise<YDisplay>
es      async get_display( )
dnp     YDisplayProxy get_display( )
cp      YDisplayProxy* get_display( )
```

Returns the parent YDisplay object of the current YDisplayLayer.

Returns :
an `YDisplay` object

displaylayer→get_displayHeight()
displaylayer→displayHeight()

YDisplayLayer

Returns the display height, in pixels.

js	function **get_displayHeight**( )
cpp	int **get_displayHeight**( )
m	-(int) displayHeight
pas	LongInt **get_displayHeight**( ): LongInt
vb	function **get_displayHeight**( ) As Integer
cs	int **get_displayHeight**( )
java	int **get_displayHeight**( )
uwp	async Task<int> **get_displayHeight**( )
py	**get_displayHeight**( )
php	function **get_displayHeight**( )
ts	async **get_displayHeight**( ): Promise<number>
es	async **get_displayHeight**( )
dnp	int **get_displayHeight**( )
cp	int **get_displayHeight**( )
cmd	YDisplay **target** [-layer **layerId**] **get_displayHeight**

Returns :
an integer corresponding to the display height, in pixels

On failure, throws an exception or returns Y_DISPLAYHEIGHT_INVALID.

displaylayer→get_displayWidth()
displaylayer→displayWidth()

YDisplayLayer

Returns the display width, in pixels.

```
js      function get_displayWidth( )
cpp     int get_displayWidth( )
m       -(int) displayWidth
pas     LongInt get_displayWidth( ): LongInt
vb      function get_displayWidth( ) As Integer
cs      int get_displayWidth( )
java    int get_displayWidth( )
uwp     async Task<int> get_displayWidth( )
py      get_displayWidth( )
php     function get_displayWidth( )
ts      async get_displayWidth( ): Promise<number>
es      async get_displayWidth( )
dnp     int get_displayWidth( )
cp      int get_displayWidth( )
cmd     YDisplay target [-layer layerId] get_displayWidth
```

Returns :
an integer corresponding to the display width, in pixels

On failure, throws an exception or returns Y_DISPLAYWIDTH_INVALID.

displaylayer→get_layerHeight() YDisplayLayer
displaylayer→layerHeight()

Returns the height of the layers to draw on, in pixels.

js	function **get_layerHeight**( )
cpp	int **get_layerHeight**( )
m	-(int) layerHeight
pas	LongInt **get_layerHeight**( ): LongInt
vb	function **get_layerHeight**( ) As Integer
cs	int **get_layerHeight**( )
java	int **get_layerHeight**( )
uwp	async Task<int> **get_layerHeight**( )
py	**get_layerHeight**( )
php	function **get_layerHeight**( )
ts	async **get_layerHeight**( ): Promise<number>
es	async **get_layerHeight**( )
dnp	int **get_layerHeight**( )
cp	int **get_layerHeight**( )
cmd	YDisplay **target** [-layer **layerId**] **get_layerHeight**

Returns :
an integer corresponding to the height of the layers to draw on, in pixels

On failure, throws an exception or returns Y_LAYERHEIGHT_INVALID.

displaylayer→get_layerWidth()
displaylayer→layerWidth()

YDisplayLayer

Returns the width of the layers to draw on, in pixels.

```
js    function get_layerWidth( )
cpp   int get_layerWidth( )
m     -(int) layerWidth
pas   LongInt get_layerWidth( ): LongInt
vb    function get_layerWidth( ) As Integer
cs    int get_layerWidth( )
java  int get_layerWidth( )
uwp   async Task<int> get_layerWidth( )
py    get_layerWidth( )
php   function get_layerWidth( )
ts    async get_layerWidth( ): Promise<number>
es    async get_layerWidth( )
dnp   int get_layerWidth( )
cp    int get_layerWidth( )
cmd   YDisplay target [-layer layerId] get_layerWidth
```

Returns :
an integer corresponding to the width of the layers to draw on, in pixels

On failure, throws an exception or returns Y_LAYERWIDTH_INVALID.

displaylayer→hide() YDisplayLayer

Hides the layer.

js	function **hide**( )
cpp	int **hide**( )
m	-(int) **hide**
pas	LongInt **hide**( ): LongInt
vb	function **hide**( ) As Integer
cs	int **hide**( )
java	int **hide**( )
uwp	async Task<int> **hide**( )
py	**hide**( )
php	function **hide**( )
ts	async **hide**( ): Promise<number>
es	async **hide**( )
dnp	int **hide**( )
cp	int **hide**( )
cmd	YDisplay **target** [-layer **layerId**] **hide**

The state of the layer is preserved but the layer is not displayed on the screen until the next call to `unhide()`. Hiding the layer can positively affect the drawing speed, since it postpones the rendering until all operations are completed (double-buffering).

Returns :
`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→lineTo() YDisplayLayer

Draws a line from current drawing pointer position to the specified position.

```
js      function lineTo( x, y)
cpp     int lineTo( int x, int y)
m       -(int) lineTo : (int) x
                      : (int) y
pas     LongInt lineTo( x: LongInt, y: LongInt): LongInt
vb      function lineTo( ByVal x As Integer, ByVal y As Integer) As Integer
cs      int lineTo( int x, int y)
java    int lineTo( int x, int y)
uwp     async Task<int> lineTo( int x, int y)
py      lineTo( x, y)
php     function lineTo( $x, $y)
ts      async lineTo( x: number, y: number): Promise<number>
es      async lineTo( x, y)
dnp     int lineTo( int x, int y)
cp      int lineTo( int x, int y)
cmd     YDisplay target [-layer layerId] lineTo  x y
```

The specified destination pixel is included in the line. The pointer position is then moved to the end point of the line.

Parameters :

x the distance from left of layer to the end point of the line, in pixels

y the distance from top of layer to the end point of the line, in pixels

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→moveTo() YDisplayLayer

Moves the drawing pointer of this layer to the specified position.

```
js    function moveTo( x, y)
cpp   int moveTo( int x, int y)
m     -(int) moveTo : (int) x
                    : (int) y
pas   LongInt moveTo( x: LongInt, y: LongInt): LongInt
vb    function moveTo( ByVal x As Integer, ByVal y As Integer) As Integer
cs    int moveTo( int x, int y)
java  int moveTo( int x, int y)
uwp   async Task<int> moveTo( int x, int y)
py    moveTo( x, y)
php   function moveTo( $x, $y)
ts    async moveTo( x: number, y: number): Promise<number>
es    async moveTo( x, y)
dnp   int moveTo( int x, int y)
cp    int moveTo( int x, int y)
cmd   YDisplay target [-layer layerId] moveTo  x y
```

Parameters :

x the distance from left of layer, in pixels

y the distance from top of layer, in pixels

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→reset() YDisplayLayer

Reverts the layer to its initial state (fully transparent, default settings).

```
js    function reset( )
cpp   int reset( )
m     -(int) reset
pas   LongInt reset( ): LongInt
vb    function reset( ) As Integer
cs    int reset( )
java  int reset( )
uwp   async Task<int> reset( )
py    reset( )
php   function reset( )
ts    async reset( ): Promise<number>
es    async reset( )
dnp   int reset( )
cp    int reset( )
cmd   YDisplay target [-layer layerId] reset
```

Reinitializes the drawing pointer to the upper left position, and selects the most visible pen color. If you only want to erase the layer content, use the method `clear()` instead.

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→selectColorPen() YDisplayLayer

Selects the pen color for all subsequent drawing functions, including text drawing.

```
js      function selectColorPen( color)
cpp     int selectColorPen( int color)
m       -(int) selectColorPen : (int) color
pas     LongInt selectColorPen( color: LongInt): LongInt
vb      function selectColorPen( ByVal color As Integer) As Integer
cs      int selectColorPen( int color)
java    int selectColorPen( int color)
uwp     async Task<int> selectColorPen( int color)
py      selectColorPen( color)
php     function selectColorPen( $color)
ts      async selectColorPen( color: number): Promise<number>
es      async selectColorPen( color)
dnp     int selectColorPen( int color)
cp      int selectColorPen( int color)
cmd     YDisplay target [-layer layerId] selectColorPen  color
```

The pen color is provided as an RGB value. For grayscale or monochrome displays, the value is automatically converted to the proper range.

Parameters :

color the desired pen color, as a 24-bit RGB value

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→selectEraser() YDisplayLayer

Selects an eraser instead of a pen for all subsequent drawing functions, except for bitmap copy functions.

```
js      function selectEraser( )
cpp     int selectEraser( )
m       -(int) selectEraser
pas     LongInt selectEraser( ): LongInt
vb      function selectEraser( ) As Integer
cs      int selectEraser( )
java    int selectEraser( )
uwp     async Task<int> selectEraser( )
py      selectEraser( )
php     function selectEraser( )
ts      async selectEraser( ): Promise<number>
es      async selectEraser( )
dnp     int selectEraser( )
cp      int selectEraser( )
cmd     YDisplay target [-layer layerId] selectEraser
```

Any point drawn using the eraser becomes transparent (as when the layer is empty), showing the other layers beneath it.

Returns :
YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→selectFont() YDisplayLayer

Selects a font to use for the next text drawing functions, by providing the name of the font file.

```
js    function selectFont( fontname)
cpp   int selectFont( string fontname)
m     -(int) selectFont : (NSString*) fontname
pas   LongInt selectFont( fontname: string): LongInt
vb    function selectFont( ByVal fontname As String) As Integer
cs    int selectFont( string fontname)
java  int selectFont( String fontname)
uwp   async Task<int> selectFont( string fontname)
py    selectFont( fontname)
php   function selectFont( $fontname)
ts    async selectFont( fontname: string): Promise<number>
es    async selectFont( fontname)
dnp   int selectFont( string fontname)
cp    int selectFont( string fontname)
cmd   YDisplay target [-layer layerId] selectFont  fontname
```

You can use a built-in font as well as a font file that you have previously uploaded to the device built-in memory. If you experience problems selecting a font file, check the device logs for any error message such as missing font file or bad font file format.

Parameters :

fontname the font file name, embedded fonts are 8x8.yfm, Small.yfm, Medium.yfm, Large.yfm (not available on Yocto-MiniDisplay).

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→selectGrayPen() YDisplayLayer

Selects the pen gray level for all subsequent drawing functions, including text drawing.

```
js    function selectGrayPen( graylevel)
cpp   int selectGrayPen( int graylevel)
m     -(int) selectGrayPen : (int) graylevel
pas   LongInt selectGrayPen( graylevel: LongInt): LongInt
vb    function selectGrayPen( ByVal graylevel As Integer) As Integer
cs    int selectGrayPen( int graylevel)
java  int selectGrayPen( int graylevel)
uwp   async Task<int> selectGrayPen( int graylevel)
py    selectGrayPen( graylevel)
php   function selectGrayPen( $graylevel)
ts    async selectGrayPen( graylevel: number): Promise<number>
es    async selectGrayPen( graylevel)
dnp   int selectGrayPen( int graylevel)
cp    int selectGrayPen( int graylevel)
cmd   YDisplay target [-layer layerId] selectGrayPen  graylevel
```

The gray level is provided as a number between 0 (black) and 255 (white, or whichever the lightest color is). For monochrome displays (without gray levels), any value lower than 128 is rendered as black, and any value equal or above to 128 is non-black.

Parameters :

graylevel the desired gray level, from 0 to 255

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setAntialiasingMode() YDisplayLayer

Enables or disables anti-aliasing for drawing oblique lines and circles.

```
js    function setAntialiasingMode( mode)
cpp   int setAntialiasingMode( bool mode)
m     -(int) setAntialiasingMode : (bool) mode
pas   LongInt setAntialiasingMode( mode: boolean): LongInt
vb    function setAntialiasingMode( ByVal mode As Boolean) As Integer
cs    int setAntialiasingMode( bool mode)
java  int setAntialiasingMode( boolean mode)
uwp   async Task<int> setAntialiasingMode( bool mode)
py    setAntialiasingMode( mode)
php   function setAntialiasingMode( $mode)
ts    async setAntialiasingMode( mode: boolean): Promise<number>
es    async setAntialiasingMode( mode)
dnp   int setAntialiasingMode( bool mode)
cp    int setAntialiasingMode( bool mode)
cmd   YDisplay target [-layer layerId] setAntialiasingMode  mode
```

Anti-aliasing provides a smoother aspect when looked from far enough, but it can add fuzziness when the display is looked from very close. At the end of the day, it is your personal choice. Anti-aliasing is enabled by default on grayscale and color displays, but you can disable it if you prefer. This setting has no effect on monochrome displays.

Parameters :

mode `true` to enable anti-aliasing, `false` to disable it.

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setConsoleBackground() YDisplayLayer

Sets up the background color used by the `clearConsole` function and by the console scrolling feature.

```
js    function setConsoleBackground( bgcol)
cpp   int setConsoleBackground( int bgcol)
m     -(int) setConsoleBackground : (int) bgcol
pas   LongInt setConsoleBackground( bgcol: LongInt): LongInt
vb    function setConsoleBackground( ByVal bgcol As Integer) As Integer
cs    int setConsoleBackground( int bgcol)
java  int setConsoleBackground( int bgcol)
uwp   async Task<int> setConsoleBackground( int bgcol)
py    setConsoleBackground( bgcol)
php   function setConsoleBackground( $bgcol)
ts    async setConsoleBackground( bgcol: number): Promise<number>
es    async setConsoleBackground( bgcol)
dnp   int setConsoleBackground( int bgcol)
cp    int setConsoleBackground( int bgcol)
cmd   YDisplay target [-layer layerId] setConsoleBackground  bgcol
```

Parameters :

bgcol the background gray level to use when scrolling (0 = black, 255 = white), or -1 for transparent

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setConsoleMargins() YDisplayLayer

Sets up display margins for the `consoleOut` function.

```
js    function setConsoleMargins( x1, y1, x2, y2)
cpp   int setConsoleMargins( int x1, int y1, int x2, int y2)
m     -(int) setConsoleMargins : (int) x1
                               : (int) y1
                               : (int) x2
                               : (int) y2
pas   LongInt setConsoleMargins( x1: LongInt,
                                 y1: LongInt,
                                 x2: LongInt,
                                 y2: LongInt): LongInt
vb    function setConsoleMargins( ByVal x1 As Integer,
                                  ByVal y1 As Integer,
                                  ByVal x2 As Integer,
                                  ByVal y2 As Integer) As Integer
cs    int setConsoleMargins( int x1, int y1, int x2, int y2)
java  int setConsoleMargins( int x1, int y1, int x2, int y2)
uwp   async Task<int> setConsoleMargins( int x1, int y1, int x2, int y2)
py    setConsoleMargins( x1, y1, x2, y2)
php   function setConsoleMargins( $x1, $y1, $x2, $y2)
ts    async setConsoleMargins( x1: number, y1: number, x2: number, y2: number): Promise<number>
es    async setConsoleMargins( x1, y1, x2, y2)
dnp   int setConsoleMargins( int x1, int y1, int x2, int y2)
cp    int setConsoleMargins( int x1, int y1, int x2, int y2)
cmd   YDisplay target [-layer layerId] setConsoleMargins  x1 y1 x2 y2
```

Parameters :

x1 the distance from left of layer to the left margin, in pixels
y1 the distance from top of layer to the top margin, in pixels
x2 the distance from left of layer to the right margin, in pixels
y2 the distance from top of layer to the bottom margin, in pixels

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setConsoleWordWrap() YDisplayLayer

Sets up the wrapping behavior used by the `consoleOut` function.

```
js    function setConsoleWordWrap( wordwrap)
cpp   int setConsoleWordWrap( bool wordwrap)
m     -(int) setConsoleWordWrap : (bool) wordwrap
pas   LongInt setConsoleWordWrap( wordwrap: boolean): LongInt
vb    function setConsoleWordWrap( ByVal wordwrap As Boolean) As Integer
cs    int setConsoleWordWrap( bool wordwrap)
java  int setConsoleWordWrap( boolean wordwrap)
uwp   async Task<int> setConsoleWordWrap( bool wordwrap)
py    setConsoleWordWrap( wordwrap)
php   function setConsoleWordWrap( $wordwrap)
ts    async setConsoleWordWrap( wordwrap: boolean): Promise<number>
es    async setConsoleWordWrap( wordwrap)
dnp   int setConsoleWordWrap( bool wordwrap)
cp    int setConsoleWordWrap( bool wordwrap)
cmd   YDisplay target [-layer layerId] setConsoleWordWrap  wordwrap
```

Parameters :

wordwrap `true` to wrap only between words, `false` to wrap on the last column anyway.

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→setLayerPosition() YDisplayLayer

Sets the position of the layer relative to the display upper left corner.

```
js      function setLayerPosition( x, y, scrollTime)
cpp     int setLayerPosition( int x, int y, int scrollTime)
m       -(int) setLayerPosition : (int) x
                                : (int) y
                                : (int) scrollTime
pas     LongInt setLayerPosition( x: LongInt,
                                  y: LongInt,
                                  scrollTime: LongInt): LongInt
vb      function setLayerPosition( ByVal x As Integer,
                                   ByVal y As Integer,
                                   ByVal scrollTime As Integer) As Integer
cs      int setLayerPosition( int x, int y, int scrollTime)
java    int setLayerPosition( int x, int y, int scrollTime)
uwp     async Task<int> setLayerPosition( int x, int y, int scrollTime)
py      setLayerPosition( x, y, scrollTime)
php     function setLayerPosition( $x, $y, $scrollTime)
ts      async setLayerPosition( x: number, y: number, scrollTime: number): Promise<number>
es      async setLayerPosition( x, y, scrollTime)
dnp     int setLayerPosition( int x, int y, int scrollTime)
cp      int setLayerPosition( int x, int y, int scrollTime)
cmd     YDisplay target [-layer layerId] setLayerPosition  x y scrollTime
```

When smooth scrolling is used, the display offset of the layer is automatically updated during the next milliseconds to animate the move of the layer.

Parameters :

x	the distance from left of display to the upper left corner of the layer
y	the distance from top of display to the upper left corner of the layer
scrollTime	number of milliseconds to use for smooth scrolling, or 0 if the scrolling should be immediate.

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

displaylayer→unhide() YDisplayLayer

Shows the layer.

```
js      function unhide( )
cpp     int unhide( )
m       -(int) unhide
pas     LongInt unhide( ): LongInt
vb      function unhide( ) As Integer
cs      int unhide( )
java    int unhide( )
uwp     async Task<int> unhide( )
py      unhide( )
php     function unhide( )
ts      async unhide( ): Promise<number>
es      async unhide( )
dnp     int unhide( )
cp      int unhide( )
cmd     YDisplay target [-layer layerId] unhide
```

Shows the layer again after a hide command.

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

24.5. Class YAnButton

Analog input control interface, available for instance in the Yocto-Buzzer, the Yocto-Knob, the Yocto-MaxiBuzzer or the Yocto-MaxiDisplay

The `YAnButton` class provide access to basic resistive inputs. Such inputs can be used to measure the state of a simple button as well as to read an analog potentiometer (variable resistance). This can be use for instance with a continuous rotating knob, a throttle grip or a joystick. The module is capable to calibrate itself on min and max values, in order to compute a calibrated value that varies proportionally with the potentiometer position, regardless of its total resistance.

In order to use the functions described here, you should include:

es	in HTML: <script src="../../lib/yocto_anbutton.js"></script> in node.js: require('yoctolib-es2017/yocto_anbutton.js');
js	<script type='text/javascript' src='yocto_anbutton.js'></script>
cpp	#include "yocto_anbutton.h"
m	#import "yocto_anbutton.h"
pas	uses yocto_anbutton;
vb	yocto_anbutton.vb
cs	yocto_anbutton.cs
java	import com.yoctopuce.YoctoAPI.YAnButton;
uwp	import com.yoctopuce.YoctoAPI.YAnButton;
py	from yocto_anbutton import *
php	require_once('yocto_anbutton.php');
ts	in HTML: import { YAnButton } from '../../dist/esm/yocto_anbutton.js'; in Node.js: import { YAnButton } from 'yoctolib-cjs/yocto_anbutton.js';
dnp	import YoctoProxyAPI.YAnButtonProxy
cp	#include "yocto_anbutton_proxy.h"
vi	YAnButton.vi
ml	import YoctoProxyAPI.YAnButtonProxy

Global functions

YAnButton.FindAnButton(**func**)

Retrieves an analog input for a given identifier.

YAnButton.FindAnButtonInContext(**yctx**, **func**)

Retrieves an analog input for a given identifier in a YAPI context.

YAnButton.FirstAnButton()

Starts the enumeration of analog inputs currently accessible.

YAnButton.FirstAnButtonInContext(**yctx**)

Starts the enumeration of analog inputs currently accessible.

YAnButton.GetSimilarFunctions()

Enumerates all functions of type AnButton available on the devices currently reachable by the library, and returns their unique hardware ID.

YAnButton properties

anbutton→AdvertisedValue *[read-only]*

Short string representing the current state of the function.

anbutton→AnalogCalibration *[writable]*

Tells if a calibration process is currently ongoing.

anbutton→CalibratedValue *[read-only]*

Current calibrated input value (between 0 and 1000, included).

anbutton→CalibrationMax *[writable]*

Maximal value measured during the calibration (between 0 and 4095, included).

anbutton→CalibrationMin *[writable]*

Minimal value measured during the calibration (between 0 and 4095, included).

anbutton→FriendlyName *[read-only]*

Global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

anbutton→FunctionId *[read-only]*

Hardware identifier of the analog input, without reference to the module.

anbutton→HardwareId *[read-only]*

Unique hardware identifier of the function in the form `SERIAL.FUNCTIONID`.

anbutton→InputType *[writable]*

Decoding method applied to the input (analog or multiplexed binary switches).

anbutton→IsOnline *[read-only]*

Checks if the function is currently reachable.

anbutton→IsPressed *[read-only]*

True if the input (considered as binary) is active (closed contact), and false otherwise.

anbutton→LogicalName *[writable]*

Logical name of the function.

anbutton→Sensitivity *[writable]*

Sensibility for the input (between 1 and 1000) for triggering user callbacks.

anbutton→SerialNumber *[read-only]*

Serial number of the module, as set by the factory.

`YAnButton` methods

anbutton→clearCache()

Invalidates the cache.

anbutton→describe()

Returns a short text that describes unambiguously the instance of the analog input in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

anbutton→get_advertisedValue()

Returns the current value of the analog input (no more than 6 characters).

anbutton→get_analogCalibration()

Tells if a calibration process is currently ongoing.

anbutton→get_calibratedValue()

Returns the current calibrated input value (between 0 and 1000, included).

anbutton→get_calibrationMax()

Returns the maximal value measured during the calibration (between 0 and 4095, included).

anbutton→get_calibrationMin()

Returns the minimal value measured during the calibration (between 0 and 4095, included).

anbutton→get_errorMessage()

Returns the error message of the latest error with the analog input.

anbutton→get_errorType()

Returns the numerical error code of the latest error with the analog input.

anbutton→get_friendlyName()

Returns a global identifier of the analog input in the format `MODULE_NAME.FUNCTION_NAME`.

anbutton→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

anbutton→get_functionId()

Returns the hardware identifier of the analog input, without reference to the module.

anbutton→get_hardwareId()

Returns the unique hardware identifier of the analog input in the form SERIAL.FUNCTIONID.

anbutton→get_inputType()

Returns the decoding method applied to the input (analog or multiplexed binary switches).

anbutton→get_isPressed()

Returns true if the input (considered as binary) is active (closed contact), and false otherwise.

anbutton→get_lastTimePressed()

Returns the number of elapsed milliseconds between the module power on and the last time the input button was pressed (the input contact transitioned from open to closed).

anbutton→get_lastTimeReleased()

Returns the number of elapsed milliseconds between the module power on and the last time the input button was released (the input contact transitioned from closed to open).

anbutton→get_logicalName()

Returns the logical name of the analog input.

anbutton→get_module()

Gets the YModule object for the device on which the function is located.

anbutton→get_module_async(**callback**, **context**)

Gets the YModule object for the device on which the function is located (asynchronous version).

anbutton→get_pulseCounter()

Returns the pulse counter value.

anbutton→get_pulseTimer()

Returns the timer of the pulses counter (ms).

anbutton→get_rawValue()

Returns the current measured input value as-is (between 0 and 4095, included).

anbutton→get_sensitivity()

Returns the sensibility for the input (between 1 and 1000) for triggering user callbacks.

anbutton→get_serialNumber()

Returns the serial number of the module, as set by the factory.

anbutton→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

anbutton→isOnline()

Checks if the analog input is currently reachable, without raising any error.

anbutton→isOnline_async(**callback**, **context**)

Checks if the analog input is currently reachable, without raising any error (asynchronous version).

anbutton→isReadOnly()

Test if the function is readOnly.

anbutton→load(**msValidity**)

Preloads the analog input cache with a specified validity duration.

anbutton→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.

anbutton→load_async(**msValidity**, **callback**, **context**)

Preloads the analog input cache with a specified validity duration (asynchronous version).

anbutton→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

anbutton→nextAnButton()

Continues the enumeration of analog inputs started using `yFirstAnButton()`.

anbutton→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

anbutton→resetCounter()

Returns the pulse counter value as well as its timer.

anbutton→set_analogCalibration(**newval**)

Starts or stops the calibration process.

anbutton→set_calibrationMax(**newval**)

Changes the maximal calibration value for the input (between 0 and 4095, included), without actually starting the automated calibration.

anbutton→set_calibrationMin(**newval**)

Changes the minimal calibration value for the input (between 0 and 4095, included), without actually starting the automated calibration.

anbutton→set_inputType(**newval**)

Changes the decoding method applied to the input (analog or multiplexed binary switches).

anbutton→set_logicalName(**newval**)

Changes the logical name of the analog input.

anbutton→set_sensitivity(**newval**)

Changes the sensibility for the input (between 1 and 1000) for triggering user callbacks.

anbutton→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

anbutton→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

anbutton→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YAnButton.FindAnButton() YAnButton
YAnButton.FindAnButton()

Retrieves an analog input for a given identifier.

```
js      function yFindAnButton( func)
cpp     YAnButton* FindAnButton( string func)
m       +(YAnButton*) FindAnButton : (NSString*) func
pas     TYAnButton yFindAnButton( func: string): TYAnButton
vb      function FindAnButton( ByVal func As String) As YAnButton
cs      static YAnButton FindAnButton( string func)
java    static YAnButton FindAnButton( String func)
uwp     static YAnButton FindAnButton( string func)
py      FindAnButton( func)
php     function FindAnButton( $func)
ts      static FindAnButton( func: string): YAnButton
es      static FindAnButton( func)
dnp     static YAnButtonProxy FindAnButton( string func)
cp      static YAnButtonProxy * FindAnButton( string func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the analog input is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YAnButton.isOnline()` to test if the analog input is indeed online at a given time. In case of ambiguity when looking for an analog input by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the analog input, for instance `YBUZZER2.anButton1`.

Returns :

a `YAnButton` object allowing you to drive the analog input.

YAnButton.FindAnButtonInContext() YAnButton
YAnButton.FindAnButtonInContext()

Retrieves an analog input for a given identifier in a YAPI context.

java static YAnButton **FindAnButtonInContext**(YAPIContext **yctx**, String **func**)
uwp static YAnButton **FindAnButtonInContext**(YAPIContext **yctx**, string **func**)
ts static **FindAnButtonInContext**(**yctx**: YAPIContext, **func**: string): YAnButton
es static **FindAnButtonInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the analog input is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YAnButton.isOnline()` to test if the analog input is indeed online at a given time. In case of ambiguity when looking for an analog input by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the analog input, for instance `YBUZZER2.anButton1`.

Returns :

a `YAnButton` object allowing you to drive the analog input.

YAnButton.FirstAnButton() YAnButton
YAnButton.FirstAnButton()

Starts the enumeration of analog inputs currently accessible.

js	function **yFirstAnButton**( )
cpp	YAnButton * **FirstAnButton**( )
m	+(YAnButton*) **FirstAnButton**
pas	TYAnButton **yFirstAnButton**( ): TYAnButton
vb	function **FirstAnButton**( ) As YAnButton
cs	static YAnButton **FirstAnButton**( )
java	static YAnButton **FirstAnButton**( )
uwp	static YAnButton **FirstAnButton**( )
py	**FirstAnButton**( )
php	function **FirstAnButton**( )
ts	static **FirstAnButton**( ): YAnButton | null
es	static **FirstAnButton**( )

Use the method `YAnButton.nextAnButton()` to iterate on next analog inputs.

Returns :

a pointer to a `YAnButton` object, corresponding to the first analog input currently online, or a `null` pointer if there are none.

YAnButton.FirstAnButtonInContext() YAnButton
YAnButton.FirstAnButtonInContext()

Starts the enumeration of analog inputs currently accessible.

```
java  static YAnButton FirstAnButtonInContext( YAPIContext yctx)
uwp   static YAnButton FirstAnButtonInContext( YAPIContext yctx)
ts    static FirstAnButtonInContext( yctx: YAPIContext): YAnButton | null
es    static FirstAnButtonInContext( yctx)
```

Use the method `YAnButton.nextAnButton()` to iterate on next analog inputs.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YAnButton` object, corresponding to the first analog input currently online, or a `null` pointer if there are none.

YAnButton.GetSimilarFunctions() YAnButton YAnButton.GetSimilarFunctions()

Enumerates all functions of type AnButton available on the devices currently reachable by the library, and returns their unique hardware ID.

dnp static new string[] **GetSimilarFunctions**( )

cp static vector<string> **GetSimilarFunctions**( )

Each of these IDs can be provided as argument to the method `YAnButton.FindAnButton` to obtain an object that can control the corresponding device.

Returns :

an array of strings, each string containing the unique hardwareId of a device function currently connected.

anbutton→AdvertisedValue — YAnButton

Short string representing the current state of the function.

dnp string **AdvertisedValue**

anbutton→AnalogCalibration — YAnButton

Tells if a calibration process is currently ongoing.

dnp int **AnalogCalibration**

Writable. Starts or stops the calibration process. Remember to call the `saveToFlash()` method of the module at the end of the calibration if the modification must be kept.

anbutton→CalibratedValue — YAnButton

Current calibrated input value (between 0 and 1000, included).

dnp int **CalibratedValue**

anbutton→CalibrationMax YAnButton

Maximal value measured during the calibration (between 0 and 4095, included).

dnp int **CalibrationMax**

Writable. Changes the maximal calibration value for the input (between 0 and 4095, included), without actually starting the automated calibration. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

anbutton→CalibrationMin — YAnButton

Minimal value measured during the calibration (between 0 and 4095, included).

dnp int **CalibrationMin**

Writable. Changes the minimal calibration value for the input (between 0 and 4095, included), without actually starting the automated calibration. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

anbutton→FriendlyName — YAnButton

Global identifier of the function in the format MODULE_NAME.FUNCTION_NAME.

dnp string **FriendlyName**

The returned string uses the logical names of the module and of the function if they are defined, otherwise the serial number of the module and the hardware identifier of the function (for example: MyCustomName.relay1)

anbutton→FunctionId YAnButton

Hardware identifier of the analog input, without reference to the module.

dnp string **FunctionId**

For example `relay1`

anbutton→HardwareId — YAnButton

Unique hardware identifier of the function in the form SERIAL.FUNCTIONID.

dnp string **HardwareId**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example RELAYL01-123456.relay1).

anbutton→InputType YAnButton

Decoding method applied to the input (analog or multiplexed binary switches).

dnp int **InputType**

Writable. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

anbutton→IsOnline — YAnButton

Checks if the function is currently reachable.

dnp bool **IsOnline**

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the function.

anbutton→IsPressed — YAnButton

True if the input (considered as binary) is active (closed contact), and false otherwise.

dnp int **IsPressed**

anbutton→LogicalName YAnButton

Logical name of the function.

dnp string **LogicalName**

Writable. You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

anbutton→Sensitivity — YAnButton

Sensibility for the input (between 1 and 1000) for triggering user callbacks.

dnp int **Sensitivity**

Writable. The sensibility is used to filter variations around a fixed value, but does not preclude the transmission of events when the input value evolves constantly in the same direction. Special case: when the value 1000 is used, the callback will only be thrown when the logical state of the input switches from pressed to released and back. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

anbutton→SerialNumber — YAnButton

Serial number of the module, as set by the factory.

dnp string **SerialNumber**

anbutton→clearCache() YAnButton

Invalidates the cache.

js	function **clearCache**( )
cpp	void **clearCache**( )
m	-(void) **clearCache**
pas	**clearCache**( )
vb	procedure **clearCache**( )
cs	void **clearCache**( )
java	void **clearCache**( )
py	**clearCache**( )
php	function **clearCache**( )
ts	async **clearCache**( ): Promise<void>
es	async **clearCache**( )

Invalidates the cache of the analog input attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

anbutton→describe() YAnButton

Returns a short text that describes unambiguously the instance of the analog input in the form TYPE(NAME)=SERIAL.FUNCTIONID.

```
js      function describe( )
cpp     string describe( )
m       -(NSString*) describe
pas     string describe( ): string
vb      function describe( ) As String
cs      string describe( )
java    String describe( )
py      describe( )
php     function describe( )
ts      async describe( ): Promise<string>
es      async describe( )
```

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :
a string that describes the analog input (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

anbutton→get_advertisedValue()
anbutton→advertisedValue()

YAnButton

Returns the current value of the analog input (no more than 6 characters).

```
js      function get_advertisedValue( )
cpp     string get_advertisedValue( )
m       -(NSString*) advertisedValue
pas     string get_advertisedValue( ): string
vb      function get_advertisedValue( ) As String
cs      string get_advertisedValue( )
java    String get_advertisedValue( )
uwp     async Task<string> get_advertisedValue( )
py      get_advertisedValue( )
php     function get_advertisedValue( )
ts      async get_advertisedValue( ): Promise<string>
es      async get_advertisedValue( )
dnp     string get_advertisedValue( )
cp      string get_advertisedValue( )
cmd     YAnButton target get_advertisedValue
```

Returns :

a string corresponding to the current value of the analog input (no more than 6 characters).

On failure, throws an exception or returns `YAnButton.ADVERTISEDVALUE_INVALID`.

anbutton→get_analogCalibration() YAnButton
anbutton→analogCalibration()

Tells if a calibration process is currently ongoing.

js	function **get_analogCalibration**( )
cpp	Y_ANALOGCALIBRATION_enum **get_analogCalibration**( )
m	-(Y_ANALOGCALIBRATION_enum) analogCalibration
pas	Integer **get_analogCalibration**( ): Integer
vb	function **get_analogCalibration**( ) As Integer
cs	int **get_analogCalibration**( )
java	int **get_analogCalibration**( )
uwp	async Task<int> **get_analogCalibration**( )
py	**get_analogCalibration**( )
php	function **get_analogCalibration**( )
ts	async **get_analogCalibration**( ): Promise<YAnButton_AnalogCalibration>
es	async **get_analogCalibration**( )
dnp	int **get_analogCalibration**( )
cp	int **get_analogCalibration**( )
cmd	YAnButton **target get_analogCalibration**

Returns :
either `YAnButton.ANALOGCALIBRATION_OFF` or `YAnButton.ANALOGCALIBRATION_ON`

On failure, throws an exception or returns `YAnButton.ANALOGCALIBRATION_INVALID`.

anbutton→get_calibratedValue()
anbutton→calibratedValue()

YAnButton

Returns the current calibrated input value (between 0 and 1000, included).

```
js    function get_calibratedValue( )
cpp   int get_calibratedValue( )
m     -(int) calibratedValue
pas   LongInt get_calibratedValue( ): LongInt
vb    function get_calibratedValue( ) As Integer
cs    int get_calibratedValue( )
java  int get_calibratedValue( )
uwp   async Task<int> get_calibratedValue( )
py    get_calibratedValue( )
php   function get_calibratedValue( )
ts    async get_calibratedValue( ): Promise<number>
es    async get_calibratedValue( )
dnp   int get_calibratedValue( )
cp    int get_calibratedValue( )
cmd   YAnButton target get_calibratedValue
```

Returns :

an integer corresponding to the current calibrated input value (between 0 and 1000, included)

On failure, throws an exception or returns `YAnButton.CALIBRATEDVALUE_INVALID`.

anbutton→get_calibrationMax() YAnButton
anbutton→calibrationMax()

Returns the maximal value measured during the calibration (between 0 and 4095, included).

```
js      function get_calibrationMax( )
cpp     int get_calibrationMax( )
m       -(int) calibrationMax
pas     LongInt get_calibrationMax( ): LongInt
vb      function get_calibrationMax( ) As Integer
cs      int get_calibrationMax( )
java    int get_calibrationMax( )
uwp     async Task<int> get_calibrationMax( )
py      get_calibrationMax( )
php     function get_calibrationMax( )
ts      async get_calibrationMax( ): Promise<number>
es      async get_calibrationMax( )
dnp     int get_calibrationMax( )
cp      int get_calibrationMax( )
cmd     YAnButton target get_calibrationMax
```

Returns :

an integer corresponding to the maximal value measured during the calibration (between 0 and 4095, included)

On failure, throws an exception or returns `YAnButton.CALIBRATIONMAX_INVALID`.

anbutton→get_calibrationMin()
anbutton→calibrationMin()

YAnButton

Returns the minimal value measured during the calibration (between 0 and 4095, included).

```
js      function get_calibrationMin( )
cpp     int get_calibrationMin( )
m       -(int) calibrationMin
pas     LongInt get_calibrationMin( ): LongInt
vb      function get_calibrationMin( ) As Integer
cs      int get_calibrationMin( )
java    int get_calibrationMin( )
uwp     async Task<int> get_calibrationMin( )
py      get_calibrationMin( )
php     function get_calibrationMin( )
ts      async get_calibrationMin( ): Promise<number>
es      async get_calibrationMin( )
dnp     int get_calibrationMin( )
cp      int get_calibrationMin( )
cmd     YAnButton target get_calibrationMin
```

Returns :

an integer corresponding to the minimal value measured during the calibration (between 0 and 4095, included)

On failure, throws an exception or returns `YAnButton.CALIBRATIONMIN_INVALID`.

anbutton→get_errorMessage() YAnButton
anbutton→errorMessage()

Returns the error message of the latest error with the analog input.

js	function **get_errorMessage**( )
cpp	string **get_errorMessage**( )
m	-(NSString*) errorMessage
pas	string **get_errorMessage**( ): string
vb	function **get_errorMessage**( ) As String
cs	string **get_errorMessage**( )
java	String **get_errorMessage**( )
py	**get_errorMessage**( )
php	function **get_errorMessage**( )
ts	**get_errorMessage**( ): string
es	**get_errorMessage**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the analog input object

anbutton→get_errorType()
anbutton→errorType()

YAnButton

Returns the numerical error code of the latest error with the analog input.

js	function **get_errorType**( )
cpp	YRETCODE **get_errorType**( )
m	-(YRETCODE) errorType
pas	YRETCODE **get_errorType**( ): YRETCODE
vb	function **get_errorType**( ) As YRETCODE
cs	YRETCODE **get_errorType**( )
java	int **get_errorType**( )
py	**get_errorType**( )
php	function **get_errorType**( )
ts	**get_errorType**( ): number
es	**get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the analog input object

anbutton→get_friendlyName()
anbutton→friendlyName()

YAnButton

Returns a global identifier of the analog input in the format `MODULE_NAME.FUNCTION_NAME`.

```
js      function get_friendlyName( )
cpp     string get_friendlyName( )
m       -(NSString*) friendlyName
cs      string get_friendlyName( )
java    String get_friendlyName( )
py      get_friendlyName( )
php     function get_friendlyName( )
ts      async get_friendlyName( ): Promise<string>
es      async get_friendlyName( )
dnp     string get_friendlyName( )
cp      string get_friendlyName( )
```

The returned string uses the logical names of the module and of the analog input if they are defined, otherwise the serial number of the module and the hardware identifier of the analog input (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the analog input using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `YAnButton.FRIENDLYNAME_INVALID`.

anbutton→get_functionDescriptor() YAnButton
anbutton→functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

js	function **get_functionDescriptor**( )
cpp	YFUN_DESCR **get_functionDescriptor**( )
m	-(YFUN_DESCR) functionDescriptor
pas	YFUN_DESCR **get_functionDescriptor**( ): YFUN_DESCR
vb	function **get_functionDescriptor**( ) As YFUN_DESCR
cs	YFUN_DESCR **get_functionDescriptor**( )
java	String **get_functionDescriptor**( )
py	**get_functionDescriptor**( )
php	function **get_functionDescriptor**( )
ts	async **get_functionDescriptor**( ): Promise<string>
es	async **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :
an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y$CLASSNAME$.FUNCTIONDESCRIPTOR_INVALID.

anbutton→get_functionId()
anbutton→functionId()

YAnButton

Returns the hardware identifier of the analog input, without reference to the module.

js	function **get_functionId**( )
cpp	string **get_functionId**( )
m	-(NSString*) functionId
vb	function **get_functionId**( ) As String
cs	string **get_functionId**( )
java	String **get_functionId**( )
py	**get_functionId**( )
php	function **get_functionId**( )
ts	async **get_functionId**( ): Promise<string>
es	async **get_functionId**( )
dnp	string **get_functionId**( )
cp	string **get_functionId**( )

For example `relay1`

Returns :
a string that identifies the analog input (ex: `relay1`)

On failure, throws an exception or returns `YAnButton.FUNCTIONID_INVALID`.

anbutton→get_hardwareId()
anbutton→hardwareId()

YAnButton

Returns the unique hardware identifier of the analog input in the form SERIAL.FUNCTIONID.

```
js      function get_hardwareId( )
cpp     string get_hardwareId( )
m       -(NSString*) hardwareId
vb      function get_hardwareId( ) As String
cs      string get_hardwareId( )
java    String get_hardwareId( )
py      get_hardwareId( )
php     function get_hardwareId( )
ts      async get_hardwareId( ): Promise<string>
es      async get_hardwareId( )
dnp     string get_hardwareId( )
cp      string get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the analog input (for example RELAYLO1-123456.relay1).

Returns :
a string that uniquely identifies the analog input (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns YAnButton.HARDWAREID_INVALID.

anbutton→get_inputType()
anbutton→inputType()

YAnButton

Returns the decoding method applied to the input (analog or multiplexed binary switches).

js	function **get_inputType**( )
cpp	Y_INPUTTYPE_enum **get_inputType**( )
m	-(Y_INPUTTYPE_enum) inputType
pas	Integer **get_inputType**( ): Integer
vb	function **get_inputType**( ) As Integer
cs	int **get_inputType**( )
java	int **get_inputType**( )
uwp	async Task<int> **get_inputType**( )
py	**get_inputType**( )
php	function **get_inputType**( )
ts	async **get_inputType**( ): Promise<YAnButton_InputType>
es	async **get_inputType**( )
dnp	int **get_inputType**( )
cp	int **get_inputType**( )
cmd	YAnButton **target get_inputType**

Returns :

a value among `YAnButton.INPUTTYPE_ANALOG_FAST`, `YAnButton.INPUTTYPE_DIGITAL4` and `YAnButton.INPUTTYPE_ANALOG_SMOOTH` corresponding to the decoding method applied to the input (analog or multiplexed binary switches)

On failure, throws an exception or returns `YAnButton.INPUTTYPE_INVALID`.

anbutton→get_isPressed()
anbutton→isPressed()

YAnButton

Returns true if the input (considered as binary) is active (closed contact), and false otherwise.

js	function **get_isPressed**( )
cpp	Y_ISPRESSED_enum **get_isPressed**( )
m	-(Y_ISPRESSED_enum) isPressed
pas	Integer **get_isPressed**( ): Integer
vb	function **get_isPressed**( ) As Integer
cs	int **get_isPressed**( )
java	int **get_isPressed**( )
uwp	async Task<int> **get_isPressed**( )
py	**get_isPressed**( )
php	function **get_isPressed**( )
ts	async **get_isPressed**( ): Promise<YAnButton_IsPressed>
es	async **get_isPressed**( )
dnp	int **get_isPressed**( )
cp	int **get_isPressed**( )
cmd	YAnButton **target get_isPressed**

Returns :

either `YAnButton.ISPRESSED_FALSE` or `YAnButton.ISPRESSED_TRUE`, according to true if the input (considered as binary) is active (closed contact), and false otherwise

On failure, throws an exception or returns `YAnButton.ISPRESSED_INVALID`.

anbutton→get_lastTimePressed() YAnButton
anbutton→lastTimePressed()

Returns the number of elapsed milliseconds between the module power on and the last time the input button was pressed (the input contact transitioned from open to closed).

js	function **get_lastTimePressed**( )
cpp	s64 **get_lastTimePressed**( )
m	-(s64) lastTimePressed
pas	int64 **get_lastTimePressed**( ): int64
vb	function **get_lastTimePressed**( ) As Long
cs	long **get_lastTimePressed**( )
java	long **get_lastTimePressed**( )
uwp	async Task<long> **get_lastTimePressed**( )
py	**get_lastTimePressed**( )
php	function **get_lastTimePressed**( )
ts	async **get_lastTimePressed**( ): Promise<number>
es	async **get_lastTimePressed**( )
dnp	long **get_lastTimePressed**( )
cp	s64 **get_lastTimePressed**( )
cmd	YAnButton **target get_lastTimePressed**

Returns :

an integer corresponding to the number of elapsed milliseconds between the module power on and the last time the input button was pressed (the input contact transitioned from open to closed)

On failure, throws an exception or returns `YAnButton.LASTTIMEPRESSED_INVALID`.

anbutton→get_lastTimeReleased() YAnButton
anbutton→lastTimeReleased()

Returns the number of elapsed milliseconds between the module power on and the last time the input button was released (the input contact transitioned from closed to open).

```
js      function get_lastTimeReleased( )
cpp     s64 get_lastTimeReleased( )
m       -(s64) lastTimeReleased
pas     int64 get_lastTimeReleased( ): int64
vb      function get_lastTimeReleased( ) As Long
cs      long get_lastTimeReleased( )
java    long get_lastTimeReleased( )
uwp     async Task<long> get_lastTimeReleased( )
py      get_lastTimeReleased( )
php     function get_lastTimeReleased( )
ts      async get_lastTimeReleased( ): Promise<number>
es      async get_lastTimeReleased( )
dnp     long get_lastTimeReleased( )
cp      s64 get_lastTimeReleased( )
cmd     YAnButton target get_lastTimeReleased
```

Returns :

an integer corresponding to the number of elapsed milliseconds between the module power on and the last time the input button was released (the input contact transitioned from closed to open)

On failure, throws an exception or returns `YAnButton.LASTTIMERELEASED_INVALID`.

anbutton→get_logicalName()
anbutton→logicalName()

YAnButton

Returns the logical name of the analog input.

js	function **get_logicalName**( )
cpp	string **get_logicalName**( )
m	-(NSString*) logicalName
pas	string **get_logicalName**( ): string
vb	function **get_logicalName**( ) As String
cs	string **get_logicalName**( )
java	String **get_logicalName**( )
uwp	async Task<string> **get_logicalName**( )
py	**get_logicalName**( )
php	function **get_logicalName**( )
ts	async **get_logicalName**( ): Promise<string>
es	async **get_logicalName**( )
dnp	string **get_logicalName**( )
cp	string **get_logicalName**( )
cmd	YAnButton **target get_logicalName**

Returns :
a string corresponding to the logical name of the analog input.

On failure, throws an exception or returns `YAnButton.LOGICALNAME_INVALID`.

anbutton→get_module()
anbutton→module()

YAnButton

Gets the YModule object for the device on which the function is located.

```
js      function get_module( )
cpp     YModule * get_module( )
m       -(YModule*) module
pas     TYModule get_module( ): TYModule
vb      function get_module( ) As YModule
cs      YModule get_module( )
java    YModule get_module( )
py      get_module( )
php     function get_module( )
ts      async get_module( ): Promise<YModule>
es      async get_module( )
dnp     YModuleProxy get_module( )
cp      YModuleProxy * get_module( )
```

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

anbutton→get_module_async()
anbutton→module_async()

YAnButton

Gets the YModule object for the device on which the function is located (asynchronous version).

```
js  function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as on-line.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking Firefox JavaScript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous JavasSript calls for more details.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

anbutton→get_pulseCounter() YAnButton
anbutton→pulseCounter()

Returns the pulse counter value.

js	function **get_pulseCounter**( )
cpp	s64 **get_pulseCounter**( )
m	-(s64) pulseCounter
pas	int64 **get_pulseCounter**( ): int64
vb	function **get_pulseCounter**( ) As Long
cs	long **get_pulseCounter**( )
java	long **get_pulseCounter**( )
uwp	async Task<long> **get_pulseCounter**( )
py	**get_pulseCounter**( )
php	function **get_pulseCounter**( )
ts	async **get_pulseCounter**( ): Promise<number>
es	async **get_pulseCounter**( )
dnp	long **get_pulseCounter**( )
cp	s64 **get_pulseCounter**( )
cmd	YAnButton **target get_pulseCounter**

The value is a 32 bit integer. In case of overflow (>=2^32), the counter will wrap. To reset the counter, just call the resetCounter() method.

Returns :
an integer corresponding to the pulse counter value

On failure, throws an exception or returns `YAnButton.PULSECOUNTER_INVALID`.

anbutton→get_pulseTimer()
anbutton→pulseTimer()

YAnButton

Returns the timer of the pulses counter (ms).

js	function **get_pulseTimer**( )
cpp	s64 **get_pulseTimer**( )
m	-(s64) pulseTimer
pas	int64 **get_pulseTimer**( ): int64
vb	function **get_pulseTimer**( ) As Long
cs	long **get_pulseTimer**( )
java	long **get_pulseTimer**( )
uwp	async Task<long> **get_pulseTimer**( )
py	**get_pulseTimer**( )
php	function **get_pulseTimer**( )
ts	async **get_pulseTimer**( ): Promise<number>
es	async **get_pulseTimer**( )
dnp	long **get_pulseTimer**( )
cp	s64 **get_pulseTimer**( )
cmd	YAnButton **target get_pulseTimer**

Returns :

an integer corresponding to the timer of the pulses counter (ms)

On failure, throws an exception or returns `YAnButton.PULSETIMER_INVALID`.

anbutton→get_rawValue()
anbutton→rawValue()

YAnButton

Returns the current measured input value as-is (between 0 and 4095, included).

```
js      function get_rawValue( )
cpp     int get_rawValue( )
m       -(int) rawValue
pas     LongInt get_rawValue( ): LongInt
vb      function get_rawValue( ) As Integer
cs      int get_rawValue( )
java    int get_rawValue( )
uwp     async Task<int> get_rawValue( )
py      get_rawValue( )
php     function get_rawValue( )
ts      async get_rawValue( ): Promise<number>
es      async get_rawValue( )
dnp     int get_rawValue( )
cp      int get_rawValue( )
cmd     YAnButton target get_rawValue
```

Returns :

an integer corresponding to the current measured input value as-is (between 0 and 4095, included)

On failure, throws an exception or returns `YAnButton.RAWVALUE_INVALID`.

anbutton→get_sensitivity()
anbutton→sensitivity()

YAnButton

Returns the sensibility for the input (between 1 and 1000) for triggering user callbacks.

js	function **get_sensitivity**( )
cpp	int **get_sensitivity**( )
m	-(int) sensitivity
pas	LongInt **get_sensitivity**( ): LongInt
vb	function **get_sensitivity**( ) As Integer
cs	int **get_sensitivity**( )
java	int **get_sensitivity**( )
uwp	async Task<int> **get_sensitivity**( )
py	**get_sensitivity**( )
php	function **get_sensitivity**( )
ts	async **get_sensitivity**( ): Promise<number>
es	async **get_sensitivity**( )
dnp	int **get_sensitivity**( )
cp	int **get_sensitivity**( )
cmd	YAnButton **target get_sensitivity**

Returns :

an integer corresponding to the sensibility for the input (between 1 and 1000) for triggering user callbacks

On failure, throws an exception or returns `YAnButton.SENSITIVITY_INVALID`.

anbutton→get_serialNumber()
anbutton→serialNumber()

YAnButton

Returns the serial number of the module, as set by the factory.

```
js      function get_serialNumber( )
cpp     string get_serialNumber( )
m       -(NSString*) serialNumber
pas     string get_serialNumber( ): string
vb      function get_serialNumber( ) As String
cs      string get_serialNumber( )
java    String get_serialNumber( )
uwp     async Task<string> get_serialNumber( )
py      get_serialNumber( )
php     function get_serialNumber( )
ts      async get_serialNumber( ): Promise<string>
es      async get_serialNumber( )
dnp     string get_serialNumber( )
cp      string get_serialNumber( )
cmd     YAnButton target get_serialNumber
```

Returns :

a string corresponding to the serial number of the module, as set by the factory.

On failure, throws an exception or returns YFunction.SERIALNUMBER_INVALID.

anbutton→get_userData()
anbutton→userData()

YAnButton

Returns the value of the userData attribute, as previously stored using method `set_userData`.

js	function **get_userData**( )
cpp	void * **get_userData**( )
m	-(id) userData
pas	Tobject **get_userData**( ): Tobject
vb	function **get_userData**( ) As Object
cs	object **get_userData**( )
java	Object **get_userData**( )
py	**get_userData**( )
php	function **get_userData**( )
ts	async **get_userData**( ): Promise<object|null>
es	async **get_userData**( )

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

anbutton→isOnline() YAnButton

Checks if the analog input is currently reachable, without raising any error.

```
js      function isOnline( )
cpp     bool isOnline( )
m       -(BOOL) isOnline
pas     boolean isOnline( ): boolean
vb      function isOnline( ) As Boolean
cs      bool isOnline( )
java    boolean isOnline( )
py      isOnline( )
php     function isOnline( )
ts      async isOnline( ): Promise<boolean>
es      async isOnline( )
dnp     bool isOnline( )
cp      bool isOnline( )
```

If there is a cached value for the analog input in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the analog input.

Returns :

`true` if the analog input can be reached, and `false` otherwise

anbutton→isOnline_async() YAnButton

Checks if the analog input is currently reachable, without raising any error (asynchronous version).

```
js  function isOnline_async( callback, context)
```

If there is a cached value for the analog input in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

anbutton→isReadOnly() YAnButton

Test if the function is readOnly.

cpp	bool **isReadOnly**( )
m	-(bool) **isReadOnly**
pas	boolean **isReadOnly**( ): boolean
vb	function **isReadOnly**( ) As Boolean
cs	bool **isReadOnly**( )
java	boolean **isReadOnly**( )
uwp	async Task<bool> **isReadOnly**( )
py	**isReadOnly**( )
php	function **isReadOnly**( )
ts	async **isReadOnly**( ): Promise<boolean>
es	async **isReadOnly**( )
dnp	bool **isReadOnly**( )
cp	bool **isReadOnly**( )
cmd	YAnButton **target isReadOnly**

Return `true` if the function is write protected or that the function is not available.

Returns :
`true` if the function is readOnly or not online.

anbutton→load() YAnButton

Preloads the analog input cache with a specified validity duration.

```
js      function load( msValidity)
cpp     YRETCODE load( int msValidity)
m       -(YRETCODE) load : (u64) msValidity
pas     YRETCODE load( msValidity: u64): YRETCODE
vb      function load( ByVal msValidity As Long) As YRETCODE
cs      YRETCODE load( ulong msValidity)
java    int load( long msValidity)
py      load( msValidity)
php     function load( $msValidity)
ts      async load( msValidity: number): Promise<number>
es      async load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI.SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

anbutton→loadAttribute() YAnButton

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
js      function loadAttribute( attrName)
cpp     string loadAttribute( string attrName)
m       -(NSString*) loadAttribute : (NSString*) attrName
pas     string loadAttribute( attrName: string): string
vb      function loadAttribute( ByVal attrName As String) As String
cs      string loadAttribute( string attrName)
java    String loadAttribute( String attrName)
uwp     async Task<string> loadAttribute( string attrName)
py      loadAttribute( attrName)
php     function loadAttribute( $attrName)
ts      async loadAttribute( attrName: string): Promise<string>
es      async loadAttribute( attrName)
dnp     string loadAttribute( string attrName)
cp      string loadAttribute( string attrName)
```

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

anbutton→load_async() YAnButton

Preloads the analog input cache with a specified validity duration (asynchronous version).

```
js  function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking the JavaScript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or YAPI.SUCCESS)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

anbutton→muteValueCallbacks() YAnButton

Disables the propagation of every new advertised value to the parent hub.

```
js      function muteValueCallbacks( )
cpp     int muteValueCallbacks( )
m       -(int) muteValueCallbacks
pas     LongInt muteValueCallbacks( ): LongInt
vb      function muteValueCallbacks( ) As Integer
cs      int muteValueCallbacks( )
java    int muteValueCallbacks( )
uwp     async Task<int> muteValueCallbacks( )
py      muteValueCallbacks( )
php     function muteValueCallbacks( )
ts      async muteValueCallbacks( ): Promise<number>
es      async muteValueCallbacks( )
dnp     int muteValueCallbacks( )
cp      int muteValueCallbacks( )
cmd     YAnButton target muteValueCallbacks
```

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI.SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

anbutton→nextAnButton() YAnButton

Continues the enumeration of analog inputs started using `yFirstAnButton()`.

js	function **nextAnButton**( )
cpp	YAnButton * **nextAnButton**( )
m	-(nullable YAnButton*) **nextAnButton**
pas	TYAnButton **nextAnButton**( ): TYAnButton
vb	function **nextAnButton**( ) As YAnButton
cs	YAnButton **nextAnButton**( )
java	YAnButton **nextAnButton**( )
uwp	YAnButton **nextAnButton**( )
py	**nextAnButton**( )
php	function **nextAnButton**( )
ts	**nextAnButton**( ): YAnButton | null
es	**nextAnButton**( )

Caution: You can't make any assumption about the returned analog inputs order. If you want to find a specific an analog input, use `AnButton.findAnButton()` and a hardwareID or a logical name.

Returns :

a pointer to a `YAnButton` object, corresponding to an analog input currently online, or a `null` pointer if there are no more analog inputs to enumerate.

anbutton→registerValueCallback() YAnButton

Registers the callback function that is invoked on every change of advertised value.

js	function **registerValueCallback**(**callback**)
cpp	int **registerValueCallback**(YAnButtonValueCallback **callback**)
m	-(int) **registerValueCallback** : (YAnButtonValueCallback _Nullable) **callback**
pas	LongInt **registerValueCallback**(**callback**: TYAnButtonValueCallback): LongInt
vb	function **registerValueCallback**(ByVal **callback** As YAnButtonValueCallback) As Integer
cs	int **registerValueCallback**(ValueCallback **callback**)
java	int **registerValueCallback**(UpdateCallback **callback**)
uwp	async Task<int> **registerValueCallback**(ValueCallback **callback**)
py	**registerValueCallback**(**callback**)
php	function **registerValueCallback**(**$callback**)
ts	async **registerValueCallback**(**callback**: YAnButtonValueCallback | null): Promise<number>
es	async **registerValueCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

anbutton→resetCounter() YAnButton

Returns the pulse counter value as well as its timer.

```
js      function resetCounter( )
cpp     int resetCounter( )
m       -(int) resetCounter
pas     LongInt resetCounter( ): LongInt
vb      function resetCounter( ) As Integer
cs      int resetCounter( )
java    int resetCounter( )
uwp     async Task<int> resetCounter( )
py      resetCounter( )
php     function resetCounter( )
ts      async resetCounter( ): Promise<number>
es      async resetCounter( )
dnp     int resetCounter( )
cp      int resetCounter( )
cmd     YAnButton target resetCounter
```

Returns :
YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

anbutton→set_analogCalibration() YAnButton
anbutton→setAnalogCalibration()

Starts or stops the calibration process.

js	function **set_analogCalibration**(**newval**)
cpp	int **set_analogCalibration**(Y_ANALOGCALIBRATION_enum **newval**)
m	-(int) setAnalogCalibration : (Y_ANALOGCALIBRATION_enum) **newval**
pas	integer **set_analogCalibration**(**newval**: Integer): integer
vb	function **set_analogCalibration**(ByVal **newval** As Integer) As Integer
cs	int **set_analogCalibration**(int **newval**)
java	int **set_analogCalibration**(int **newval**)
uwp	async Task<int> **set_analogCalibration**(int **newval**)
py	**set_analogCalibration**(**newval**)
php	function **set_analogCalibration**(**$newval**)
ts	async **set_analogCalibration**(**newval**: YAnButton_AnalogCalibration): Promise<number>
es	async **set_analogCalibration**(**newval**)
dnp	int **set_analogCalibration**(int **newval**)
cp	int **set_analogCalibration**(int **newval**)
cmd	YAnButton **target set_analogCalibration newval**

Remember to call the `saveToFlash()` method of the module at the end of the calibration if the modification must be kept.

Parameters :

newval either `YAnButton.ANALOGCALIBRATION_OFF` or `YAnButton.ANALOGCALIBRATION_ON`

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

anbutton→set_calibrationMax() YAnButton
anbutton→setCalibrationMax()

Changes the maximal calibration value for the input (between 0 and 4095, included), without actually starting the automated calibration.

js	function **set_calibrationMax**(**newval**)
cpp	int **set_calibrationMax**(int **newval**)
m	-(int) setCalibrationMax : (int) **newval**
pas	integer **set_calibrationMax**(**newval**: LongInt): integer
vb	function **set_calibrationMax**(ByVal **newval** As Integer) As Integer
cs	int **set_calibrationMax**(int **newval**)
java	int **set_calibrationMax**(int **newval**)
uwp	async Task<int> **set_calibrationMax**(int **newval**)
py	**set_calibrationMax**(**newval**)
php	function **set_calibrationMax**(**$newval**)
ts	async **set_calibrationMax**(**newval**: number): Promise<number>
es	async **set_calibrationMax**(**newval**)
dnp	int **set_calibrationMax**(int **newval**)
cp	int **set_calibrationMax**(int **newval**)
cmd	YAnButton **target set_calibrationMax newval**

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer corresponding to the maximal calibration value for the input (between 0 and 4095, included), without actually starting the automated calibration

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

anbutton→set_calibrationMin()
anbutton→setCalibrationMin()

YAnButton

Changes the minimal calibration value for the input (between 0 and 4095, included), without actually starting the automated calibration.

```
js    function set_calibrationMin( newval)
cpp   int set_calibrationMin( int newval)
m     -(int) setCalibrationMin : (int) newval
pas   integer set_calibrationMin( newval: LongInt): integer
vb    function set_calibrationMin( ByVal newval As Integer) As Integer
cs    int set_calibrationMin( int newval)
java  int set_calibrationMin( int newval)
uwp   async Task<int> set_calibrationMin( int newval)
py    set_calibrationMin( newval)
php   function set_calibrationMin( $newval)
ts    async set_calibrationMin( newval: number): Promise<number>
es    async set_calibrationMin( newval)
dnp   int set_calibrationMin( int newval)
cp    int set_calibrationMin( int newval)
cmd   YAnButton target set_calibrationMin newval
```

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer corresponding to the minimal calibration value for the input (between 0 and 4095, included), without actually starting the automated calibration

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

anbutton→set_inputType()
anbutton→setInputType()

YAnButton

Changes the decoding method applied to the input (analog or multiplexed binary switches).

js	function **set_inputType**(**newval**)
cpp	int **set_inputType**(Y_INPUTTYPE_enum **newval**)
m	-(int) setInputType : (Y_INPUTTYPE_enum) **newval**
pas	integer **set_inputType**(**newval**: Integer): integer
vb	function **set_inputType**(ByVal **newval** As Integer) As Integer
cs	int **set_inputType**(int **newval**)
java	int **set_inputType**(int **newval**)
uwp	async Task<int> **set_inputType**(int **newval**)
py	**set_inputType**(**newval**)
php	function **set_inputType**(**$newval**)
ts	async **set_inputType**(**newval**: YAnButton_InputType): Promise<number>
es	async **set_inputType**(**newval**)
dnp	int **set_inputType**(int **newval**)
cp	int **set_inputType**(int **newval**)
cmd	YAnButton **target set_inputType newval**

Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a value among `YAnButton.INPUTTYPE_ANALOG_FAST`, `YAnButton.INPUTTYPE_DIGITAL4` and `YAnButton.INPUTTYPE_ANALOG_SMOOTH` corresponding to the decoding method applied to the input (analog or multiplexed binary switches)

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

anbutton→set_logicalName()
anbutton→setLogicalName()

YAnButton

Changes the logical name of the analog input.

js	function **set_logicalName**(**newval**)
cpp	int **set_logicalName**(string **newval**)
m	-(int) setLogicalName : (NSString*) **newval**
pas	integer **set_logicalName**(**newval**: string): integer
vb	function **set_logicalName**(ByVal **newval** As String) As Integer
cs	int **set_logicalName**(string **newval**)
java	int **set_logicalName**(String **newval**)
uwp	async Task<int> **set_logicalName**(string **newval**)
py	**set_logicalName**(**newval**)
php	function **set_logicalName**(**$newval**)
ts	async **set_logicalName**(**newval**: string): Promise<number>
es	async **set_logicalName**(**newval**)
dnp	int **set_logicalName**(string **newval**)
cp	int **set_logicalName**(string **newval**)
cmd	YAnButton **target set_logicalName newval**

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the analog input.

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

anbutton→set_sensitivity() YAnButton
anbutton→setSensitivity()

Changes the sensibility for the input (between 1 and 1000) for triggering user callbacks.

```
js      function set_sensitivity( newval)
cpp     int set_sensitivity( int newval)
m       -(int) setSensitivity : (int) newval
pas     integer set_sensitivity( newval: LongInt): integer
vb      function set_sensitivity( ByVal newval As Integer) As Integer
cs      int set_sensitivity( int newval)
java    int set_sensitivity( int newval)
uwp     async Task<int> set_sensitivity( int newval)
py      set_sensitivity( newval)
php     function set_sensitivity( $newval)
ts      async set_sensitivity( newval: number): Promise<number>
es      async set_sensitivity( newval)
dnp     int set_sensitivity( int newval)
cp      int set_sensitivity( int newval)
cmd     YAnButton target set_sensitivity newval
```

The sensibility is used to filter variations around a fixed value, but does not preclude the transmission of events when the input value evolves constantly in the same direction. Special case: when the value 1000 is used, the callback will only be thrown when the logical state of the input switches from pressed to released and back. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval an integer corresponding to the sensibility for the input (between 1 and 1000) for triggering user callbacks

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

anbutton→set_userData()
anbutton→setUserData()

YAnButton

Stores a user context provided as argument in the userData attribute of the function.

```
js    function set_userData( data)
cpp   void set_userData( void * data)
m     -(void) setUserData : (id) data
pas   set_userData( data: Tobject)
vb    procedure set_userData( ByVal data As Object)
cs    void set_userData( object data)
java  void set_userData( Object data)
py    set_userData( data)
php   function set_userData( $data)
ts    async set_userData( data: object|null): Promise<void>
es    async set_userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

anbutton→unmuteValueCallbacks() YAnButton

Re-enables the propagation of every new advertised value to the parent hub.

js	function **unmuteValueCallbacks**( )
cpp	int **unmuteValueCallbacks**( )
m	-(int) **unmuteValueCallbacks**
pas	LongInt **unmuteValueCallbacks**( ): LongInt
vb	function **unmuteValueCallbacks**( ) As Integer
cs	int **unmuteValueCallbacks**( )
java	int **unmuteValueCallbacks**( )
uwp	async Task<int> **unmuteValueCallbacks**( )
py	**unmuteValueCallbacks**( )
php	function **unmuteValueCallbacks**( )
ts	async **unmuteValueCallbacks**( ): Promise<number>
es	async **unmuteValueCallbacks**( )
dnp	int **unmuteValueCallbacks**( )
cp	int **unmuteValueCallbacks**( )
cmd	YAnButton **target unmuteValueCallbacks**

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI.SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

anbutton→wait_async() YAnButton

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js  function wait_async( callback, context)
ts  wait_async( callback: Function, context: object)
es  wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

24.6. Class YFiles

Filesystem control interface, available for instance in the Yocto-Color-V2, the Yocto-Serial, the YoctoHub-Ethernet or the YoctoHub-Wireless-n

The YFiles class is used to access the filesystem embedded on some Yoctopuce devices. This filesystem makes it possible for instance to design a custom web UI (for networked devices) or to add fonts (on display devices).

In order to use the functions described here, you should include:

js	<script type='text/javascript' src='yocto_files.js'></script>
cpp	#include "yocto_files.h"
m	#import "yocto_files.h"
pas	uses yocto_files;
vb	yocto_files.vb
cs	yocto_files.cs
java	import com.yoctopuce.YoctoAPI.YFiles;
uwp	import com.yoctopuce.YoctoAPI.YFiles;
py	from yocto_files import *
php	require_once('yocto_files.php');
ts	in HTML: import { YFiles } from '../../dist/esm/yocto_files.js'; in Node.js: import { YFiles } from 'yoctolib-cjs/yocto_files.js';
es	in HTML: <script src="../../lib/yocto_files.js"></script> in node.js: require('yoctolib-es2017/yocto_files.js');
dnp	import YoctoProxyAPI.YFilesProxy
cp	#include "yocto_files_proxy.h"
vi	YFiles.vi
ml	import YoctoProxyAPI.YFilesProxy

Global functions

YFiles.FindFiles(**func**)

Retrieves a filesystem for a given identifier.

YFiles.FindFilesInContext(**yctx**, **func**)

Retrieves a filesystem for a given identifier in a YAPI context.

YFiles.FirstFiles()

Starts the enumeration of filesystems currently accessible.

YFiles.FirstFilesInContext(**yctx**)

Starts the enumeration of filesystems currently accessible.

YFiles.GetSimilarFunctions()

Enumerates all functions of type Files available on the devices currently reachable by the library, and returns their unique hardware ID.

YFiles properties

files→AdvertisedValue *[read-only]*

Short string representing the current state of the function.

files→FilesCount *[read-only]*

Number of files currently loaded in the filesystem.

files→FriendlyName *[read-only]*

Global identifier of the function in the format `MODULE_NAME.FUNCTION_NAME`.

files→FunctionId *[read-only]*

Hardware identifier of the filesystem, without reference to the module.

files→HardwareId *[read-only]*

Unique hardware identifier of the function in the form SERIAL.FUNCTIONID.

files→IsOnline *[read-only]*

Checks if the function is currently reachable.

files→LogicalName *[writable]*

Logical name of the function.

files→SerialNumber *[read-only]*

Serial number of the module, as set by the factory.

YFiles methods

files→clearCache()

Invalidates the cache.

files→describe()

Returns a short text that describes unambiguously the instance of the filesystem in the form TYPE(NAME)=SERIAL.FUNCTIONID.

files→download(**pathname**)

Downloads the requested file and returns a binary buffer with its content.

files→download_async(**pathname**, **callback**, **context**)

Downloads the requested file and returns a binary buffer with its content.

files→fileExist(**filename**)

Test if a file exist on the filesystem of the module.

files→format_fs()

Reinitialize the filesystem to its clean, unfragmented, empty state.

files→get_advertisedValue()

Returns the current value of the filesystem (no more than 6 characters).

files→get_errorMessage()

Returns the error message of the latest error with the filesystem.

files→get_errorType()

Returns the numerical error code of the latest error with the filesystem.

files→get_filesCount()

Returns the number of files currently loaded in the filesystem.

files→get_freeSpace()

Returns the free space for uploading new files to the filesystem, in bytes.

files→get_friendlyName()

Returns a global identifier of the filesystem in the format MODULE_NAME.FUNCTION_NAME.

files→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

files→get_functionId()

Returns the hardware identifier of the filesystem, without reference to the module.

files→get_hardwareId()

Returns the unique hardware identifier of the filesystem in the form SERIAL.FUNCTIONID.

files→get_list(**pattern**)

Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.

files→get_logicalName()

Returns the logical name of the filesystem.

files→get_module()

Gets the `YModule` object for the device on which the function is located.

files→get_module_async(**callback**, **context**)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

files→get_serialNumber()

Returns the serial number of the module, as set by the factory.

files→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

files→isOnline()

Checks if the filesystem is currently reachable, without raising any error.

files→isOnline_async(**callback**, **context**)

Checks if the filesystem is currently reachable, without raising any error (asynchronous version).

files→isReadOnly()

Test if the function is readOnly.

files→load(**msValidity**)

Preloads the filesystem cache with a specified validity duration.

files→loadAttribute(**attrName**)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

files→load_async(**msValidity**, **callback**, **context**)

Preloads the filesystem cache with a specified validity duration (asynchronous version).

files→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

files→nextFiles()

Continues the enumeration of filesystems started using `yFirstFiles()`.

files→registerValueCallback(**callback**)

Registers the callback function that is invoked on every change of advertised value.

files→remove(**pathname**)

Deletes a file, given by its full path name, from the filesystem.

files→set_logicalName(**newval**)

Changes the logical name of the filesystem.

files→set_userData(**data**)

Stores a user context provided as argument in the userData attribute of the function.

files→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

files→upload(**pathname**, **content**)

Uploads a file to the filesystem, to the specified full path name.

files→wait_async(**callback**, **context**)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

YFiles.FindFiles() YFiles
YFiles.FindFiles()

Retrieves a filesystem for a given identifier.

```
js      function yFindFiles( func)
cpp     YFiles* FindFiles( string func)
m       +(YFiles*) FindFiles : (NSString*) func
pas     TYFiles yFindFiles( func: string): TYFiles
vb      function FindFiles( ByVal func As String) As YFiles
cs      static YFiles FindFiles( string func)
java    static YFiles FindFiles( String func)
uwp     static YFiles FindFiles( string func)
py      FindFiles( func)
php     function FindFiles( $func)
ts      static FindFiles( func: string): YFiles
es      static FindFiles( func)
dnp     static YFilesProxy FindFiles( string func)
cp      static YFilesProxy * FindFiles( string func)
```

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the filesystem is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YFiles.isOnline()` to test if the filesystem is indeed online at a given time. In case of ambiguity when looking for a filesystem by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

If a call to this object's is_online() method returns FALSE although you are certain that the matching device is plugged, make sure that you did call registerHub() at application initialization time.

Parameters :

func a string that uniquely characterizes the filesystem, for instance `YRGBLED2.files`.

Returns :

a `YFiles` object allowing you to drive the filesystem.

YFiles.FindFilesInContext() YFiles
YFiles.FindFilesInContext()

Retrieves a filesystem for a given identifier in a YAPI context.

java static YFiles **FindFilesInContext**(YAPIContext **yctx**, String **func**)
uwp static YFiles **FindFilesInContext**(YAPIContext **yctx**, string **func**)
ts static **FindFilesInContext**(**yctx**: YAPIContext, **func**: string): YFiles
es static **FindFilesInContext**(**yctx**, **func**)

The identifier can be specified using several formats:

- FunctionLogicalName
- ModuleSerialNumber.FunctionIdentifier
- ModuleSerialNumber.FunctionLogicalName
- ModuleLogicalName.FunctionIdentifier
- ModuleLogicalName.FunctionLogicalName

This function does not require that the filesystem is online at the time it is invoked. The returned object is nevertheless valid. Use the method `YFiles.isOnline()` to test if the filesystem is indeed online at a given time. In case of ambiguity when looking for a filesystem by logical name, no error is notified: the first instance found is returned. The search is performed first by hardware name, then by logical name.

Parameters :

yctx a YAPI context

func a string that uniquely characterizes the filesystem, for instance `YRGBLED2.files`.

Returns :

a `YFiles` object allowing you to drive the filesystem.

YFiles.FirstFiles() YFiles
YFiles.FirstFiles()

Starts the enumeration of filesystems currently accessible.

js	function **yFirstFiles**( )
cpp	YFiles * **FirstFiles**( )
m	+(YFiles*) **FirstFiles**
pas	TYFiles **yFirstFiles**( ): TYFiles
vb	function **FirstFiles**( ) As YFiles
cs	static YFiles **FirstFiles**( )
java	static YFiles **FirstFiles**( )
uwp	static YFiles **FirstFiles**( )
py	**FirstFiles**( )
php	function **FirstFiles**( )
ts	static **FirstFiles**( ): YFiles | null
es	static **FirstFiles**( )

Use the method `YFiles.nextFiles()` to iterate on next filesystems.

Returns :

a pointer to a `YFiles` object, corresponding to the first filesystem currently online, or a `null` pointer if there are none.

YFiles.FirstFilesInContext() YFiles
YFiles.FirstFilesInContext()

Starts the enumeration of filesystems currently accessible.

java static YFiles **FirstFilesInContext**(YAPIContext **yctx**)
uwp static YFiles **FirstFilesInContext**(YAPIContext **yctx**)
ts static **FirstFilesInContext**(**yctx**: YAPIContext): YFiles | null
es static **FirstFilesInContext**(**yctx**)

Use the method `YFiles.nextFiles()` to iterate on next filesystems.

Parameters :

yctx a YAPI context.

Returns :

a pointer to a `YFiles` object, corresponding to the first filesystem currently online, or a `null` pointer if there are none.

YFiles.GetSimilarFunctions() YFiles
YFiles.GetSimilarFunctions()

Enumerates all functions of type Files available on the devices currently reachable by the library, and returns their unique hardware ID.

dnp static new string[] **GetSimilarFunctions**( )

cp static vector<string> **GetSimilarFunctions**( )

Each of these IDs can be provided as argument to the method `YFiles.FindFiles` to obtain an object that can control the corresponding device.

Returns :

an array of strings, each string containing the unique hardwareId of a device function currently connected.

files→AdvertisedValue — YFiles

Short string representing the current state of the function.

dnp string **AdvertisedValue**

files→FilesCount — YFiles

Number of files currently loaded in the filesystem.

dnp int **FilesCount**

files→FriendlyName YFiles

Global identifier of the function in the format MODULE_NAME.FUNCTION_NAME.

dnp string **FriendlyName**

The returned string uses the logical names of the module and of the function if they are defined, otherwise the serial number of the module and the hardware identifier of the function (for example: MyCustomName.relay1)

files→FunctionId YFiles

Hardware identifier of the filesystem, without reference to the module.

dnp string **FunctionId**

For example `relay1`

files→HardwareId — YFiles

Unique hardware identifier of the function in the form SERIAL.FUNCTIONID.

dnp string **HardwareId**

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the function (for example RELAYLO1-123456.relay1).

files→IsOnline YFiles

Checks if the function is currently reachable.

dnp bool **IsOnline**

If there is a cached value for the function in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the function.

files→LogicalName YFiles

Logical name of the function.

dnp string **LogicalName**

Writable. You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

files→SerialNumber — YFiles

Serial number of the module, as set by the factory.

dnp string **SerialNumber**

files→clearCache() YFiles

Invalidates the cache.

js	function **clearCache**( )
cpp	void **clearCache**( )
m	-(void) **clearCache**
pas	**clearCache**( )
vb	procedure **clearCache**( )
cs	void **clearCache**( )
java	void **clearCache**( )
py	**clearCache**( )
php	function **clearCache**( )
ts	async **clearCache**( ): Promise<void>
es	async **clearCache**( )

Invalidates the cache of the filesystem attributes. Forces the next call to get_xxx() or loadxxx() to use values that come from the device.

files→describe() YFiles

Returns a short text that describes unambiguously the instance of the filesystem in the form TYPE(NAME)=SERIAL.FUNCTIONID.

```
js      function describe( )
cpp     string describe( )
m       -(NSString*) describe
pas     string describe( ): string
vb      function describe( ) As String
cs      string describe( )
java    String describe( )
py      describe( )
php     function describe( )
ts      async describe( ): Promise<string>
es      async describe( )
```

More precisely, TYPE is the type of the function, NAME it the name used for the first access to the function, SERIAL is the serial number of the module if the module is connected or "unresolved", and FUNCTIONID is the hardware identifier of the function if the module is connected. For example, this method returns Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1 if the module is already connected or Relay(BadCustomeName.relay1)=unresolved if the module has not yet been connected. This method does not trigger any USB or TCP transaction and can therefore be used in a debugger.

Returns :
a string that describes the filesystem (ex: Relay(MyCustomName.relay1)=RELAYLO1-123456.relay1)

files→download()

YFiles

Downloads the requested file and returns a binary buffer with its content.

```
js      function download( pathname)
cpp     string download( string pathname)
m       -(NSMutableData*) download : (NSString*) pathname
pas     TByteArray download( pathname: string): TByteArray
vb      function download( ByVal pathname As String) As Byte
cs      byte[] download( string pathname)
java    byte[] download( String pathname)
uwp     async Task<byte[]> download( string pathname)
py      download( pathname)
php     function download( $pathname)
ts      async download( pathname: string): Promise<Uint8Array>
es      async download( pathname)
dnp     byte[] download( string pathname)
cp      string download( string pathname)
cmd     YFiles target download pathname
```

Parameters :

pathname path and name of the file to download

Returns :

a binary buffer with the file content

On failure, throws an exception or returns an empty content.

files→download_async() YFiles

Downloads the requested file and returns a binary buffer with its content.

```
js  function download_async( pathname, callback, context)
```

This is the asynchronous version that uses a callback to pass the result when the download is completed.

Parameters :

pathname	path and name of the new file to load
callback	callback function that is invoked when the w The callback function receives three arguments: - the user-specific context object - the YFiles object whose download_async was invoked - a binary buffer with the file content
context	user-specific object that is passed as-is to the callback function

Returns :

nothing.

files→fileExist() YFiles

Test if a file exist on the filesystem of the module.

```
js      function fileExist( filename)
cpp     bool fileExist( string filename)
m       -(bool) fileExist : (NSString*) filename
pas     boolean fileExist( filename: string): boolean
vb      function fileExist( ByVal filename As String) As Boolean
cs      bool fileExist( string filename)
java    boolean fileExist( String filename)
uwp     async Task<bool> fileExist( string filename)
py      fileExist( filename)
php     function fileExist( $filename)
ts      async fileExist( filename: string): Promise<boolean>
es      async fileExist( filename)
dnp     bool fileExist( string filename)
cp      bool fileExist( string filename)
cmd     YFiles target fileExist filename
```

Parameters :

filename the file name to test.

Returns :

a true if the file exist, false otherwise.

On failure, throws an exception.

files→format_fs() YFiles

Reinitialize the filesystem to its clean, unfragmented, empty state.

```
js      function format_fs( )
cpp     int format_fs( )
m       -(int) format_fs
pas     LongInt format_fs( ): LongInt
vb      function format_fs( ) As Integer
cs      int format_fs( )
java    int format_fs( )
uwp     async Task<int> format_fs( )
py      format_fs( )
php     function format_fs( )
ts      async format_fs( ): Promise<number>
es      async format_fs( )
dnp     int format_fs( )
cp      int format_fs( )
cmd     YFiles target format_fs
```

All files previously uploaded are permanently lost.

Returns :
YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→get_advertisedValue()
files→advertisedValue()

YFiles

Returns the current value of the filesystem (no more than 6 characters).

js	function **get_advertisedValue**( )
cpp	string **get_advertisedValue**( )
m	-(NSString*) advertisedValue
pas	string **get_advertisedValue**( ): string
vb	function **get_advertisedValue**( ) As String
cs	string **get_advertisedValue**( )
java	String **get_advertisedValue**( )
uwp	async Task<string> **get_advertisedValue**( )
py	**get_advertisedValue**( )
php	function **get_advertisedValue**( )
ts	async **get_advertisedValue**( ): Promise<string>
es	async **get_advertisedValue**( )
dnp	string **get_advertisedValue**( )
cp	string **get_advertisedValue**( )
cmd	YFiles **target get_advertisedValue**

Returns :

a string corresponding to the current value of the filesystem (no more than 6 characters).

On failure, throws an exception or returns `YFiles.ADVERTISEDVALUE_INVALID`.

files→get_errorMessage()
files→errorMessage()

YFiles

Returns the error message of the latest error with the filesystem.

```
js      function get_errorMessage( )
cpp     string get_errorMessage( )
m       -(NSString*) errorMessage
pas     string get_errorMessage( ): string
vb      function get_errorMessage( ) As String
cs      string get_errorMessage( )
java    String get_errorMessage( )
py      get_errorMessage( )
php     function get_errorMessage( )
ts      get_errorMessage( ): string
es      get_errorMessage( )
```

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a string corresponding to the latest error message that occured while using the filesystem object

files→get_errorType() YFiles
files→errorType()

Returns the numerical error code of the latest error with the filesystem.

js	function **get_errorType**( )
cpp	YRETCODE **get_errorType**( )
m	-(YRETCODE) errorType
pas	YRETCODE **get_errorType**( ): YRETCODE
vb	function **get_errorType**( ) As YRETCODE
cs	YRETCODE **get_errorType**( )
java	int **get_errorType**( )
py	**get_errorType**( )
php	function **get_errorType**( )
ts	**get_errorType**( ): number
es	**get_errorType**( )

This method is mostly useful when using the Yoctopuce library with exceptions disabled.

Returns :
a number corresponding to the code of the latest error that occurred while using the filesystem object

files→get_filesCount()
files→filesCount()

YFiles

Returns the number of files currently loaded in the filesystem.

```
js      function get_filesCount( )
cpp     int get_filesCount( )
m       -(int) filesCount
pas     LongInt get_filesCount( ): LongInt
vb      function get_filesCount( ) As Integer
cs      int get_filesCount( )
java    int get_filesCount( )
uwp     async Task<int> get_filesCount( )
py      get_filesCount( )
php     function get_filesCount( )
ts      async get_filesCount( ): Promise<number>
es      async get_filesCount( )
dnp     int get_filesCount( )
cp      int get_filesCount( )
cmd     YFiles target get_filesCount
```

Returns :
an integer corresponding to the number of files currently loaded in the filesystem

On failure, throws an exception or returns `YFiles.FILESCOUNT_INVALID`.

files→get_freeSpace()
files→freeSpace()

YFiles

Returns the free space for uploading new files to the filesystem, in bytes.

```
js      function get_freeSpace( )
cpp     int get_freeSpace( )
m       -(int) freeSpace
pas     LongInt get_freeSpace( ): LongInt
vb      function get_freeSpace( ) As Integer
cs      int get_freeSpace( )
java    int get_freeSpace( )
uwp     async Task<int> get_freeSpace( )
py      get_freeSpace( )
php     function get_freeSpace( )
ts      async get_freeSpace( ): Promise<number>
es      async get_freeSpace( )
dnp     int get_freeSpace( )
cp      int get_freeSpace( )
cmd     YFiles target get_freeSpace
```

Returns :

an integer corresponding to the free space for uploading new files to the filesystem, in bytes

On failure, throws an exception or returns `YFiles.FREESPACE_INVALID`.

files→get_friendlyName()
files→friendlyName()

YFiles

Returns a global identifier of the filesystem in the format `MODULE_NAME.FUNCTION_NAME`.

```
js      function get_friendlyName( )
cpp     string get_friendlyName( )
m       -(NSString*) friendlyName
cs      string get_friendlyName( )
java    String get_friendlyName( )
py      get_friendlyName( )
php     function get_friendlyName( )
ts      async get_friendlyName( ): Promise<string>
es      async get_friendlyName( )
dnp     string get_friendlyName( )
cp      string get_friendlyName( )
```

The returned string uses the logical names of the module and of the filesystem if they are defined, otherwise the serial number of the module and the hardware identifier of the filesystem (for example: `MyCustomName.relay1`)

Returns :

a string that uniquely identifies the filesystem using logical names (ex: `MyCustomName.relay1`)

On failure, throws an exception or returns `YFiles.FRIENDLYNAME_INVALID`.

files→get_functionDescriptor() YFiles
files→functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

js	function **get_functionDescriptor**( )
cpp	YFUN_DESCR **get_functionDescriptor**( )
m	-(YFUN_DESCR) functionDescriptor
pas	YFUN_DESCR **get_functionDescriptor**( ): YFUN_DESCR
vb	function **get_functionDescriptor**( ) As YFUN_DESCR
cs	YFUN_DESCR **get_functionDescriptor**( )
java	String **get_functionDescriptor**( )
py	**get_functionDescriptor**( )
php	function **get_functionDescriptor**( )
ts	async **get_functionDescriptor**( ): Promise<string>
es	async **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :
an identifier of type YFUN_DESCR.

If the function has never been contacted, the returned value is Y$CLASSNAME$.FUNCTIONDESCRIPTOR_INVALID.

files→get_functionId()
files→functionId()

YFiles

Returns the hardware identifier of the filesystem, without reference to the module.

```
js      function get_functionId( )
cpp     string get_functionId( )
m       -(NSString*) functionId
vb      function get_functionId( ) As String
cs      string get_functionId( )
java    String get_functionId( )
py      get_functionId( )
php     function get_functionId( )
ts      async get_functionId( ): Promise<string>
es      async get_functionId( )
dnp     string get_functionId( )
cp      string get_functionId( )
```

For example `relay1`

Returns :
a string that identifies the filesystem (ex: `relay1`)

On failure, throws an exception or returns `YFiles.FUNCTIONID_INVALID`.

files→get_hardwareId() YFiles
files→hardwareId()

Returns the unique hardware identifier of the filesystem in the form SERIAL.FUNCTIONID.

js	function **get_hardwareId**( )
cpp	string **get_hardwareId**( )
m	-(NSString*) hardwareId
vb	function **get_hardwareId**( ) As String
cs	string **get_hardwareId**( )
java	String **get_hardwareId**( )
py	**get_hardwareId**( )
php	function **get_hardwareId**( )
ts	async **get_hardwareId**( ): Promise<string>
es	async **get_hardwareId**( )
dnp	string **get_hardwareId**( )
cp	string **get_hardwareId**( )

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the filesystem (for example RELAYLO1-123456.relay1).

Returns :

a string that uniquely identifies the filesystem (ex: RELAYLO1-123456.relay1)

On failure, throws an exception or returns YFiles.HARDWAREID_INVALID.

files→get_list()
files→list()

YFiles

Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.

```
js      function get_list( pattern)
cpp     vector<YFileRecord> get_list( string pattern)
m       -(NSMutableArray*) list : (NSString*) pattern
pas     TYFileRecordArray get_list( pattern: string): TYFileRecordArray
vb      function get_list( ByVal pattern As String) As List
cs      List<YFileRecord> get_list( string pattern)
java    ArrayList<YFileRecord> get_list( String pattern)
uwp     async Task<List<YFileRecord>> get_list( string pattern)
py      get_list( pattern)
php     function get_list( $pattern)
ts      async get_list( pattern: string): Promise<YFileRecord[]
es      async get_list( pattern)
dnp     YFileRecordProxy[] get_list( string pattern)
cp      vector<YFileRecordProxy> get_list( string pattern)
cmd     YFiles target get_list pattern
```

Parameters :

pattern an optional filter pattern, using star and question marks as wild cards. When an empty pattern is provided, all file records are returned.

Returns :

a list of `YFileRecord` objects, containing the file path and name, byte size and 32-bit CRC of the file content.

On failure, throws an exception or returns an empty list.

files→get_logicalName()
files→logicalName()

YFiles

Returns the logical name of the filesystem.

js	function **get_logicalName**( )
cpp	string **get_logicalName**( )
m	-(NSString*) logicalName
pas	string **get_logicalName**( ): string
vb	function **get_logicalName**( ) As String
cs	string **get_logicalName**( )
java	String **get_logicalName**( )
uwp	async Task<string> **get_logicalName**( )
py	**get_logicalName**( )
php	function **get_logicalName**( )
ts	async **get_logicalName**( ): Promise<string>
es	async **get_logicalName**( )
dnp	string **get_logicalName**( )
cp	string **get_logicalName**( )
cmd	YFiles **target get_logicalName**

Returns :
a string corresponding to the logical name of the filesystem.

On failure, throws an exception or returns `YFiles.LOGICALNAME_INVALID`.

files→get_module()
files→module()

YFiles

Gets the YModule object for the device on which the function is located.

js	function **get_module**( )
cpp	YModule * **get_module**( )
m	-(YModule*) module
pas	TYModule **get_module**( ): TYModule
vb	function **get_module**( ) As YModule
cs	YModule **get_module**( )
java	YModule **get_module**( )
py	**get_module**( )
php	function **get_module**( )
ts	async **get_module**( ): Promise<YModule>
es	async **get_module**( )
dnp	YModuleProxy **get_module**( )
cp	YModuleProxy * **get_module**( )

If the function cannot be located on any module, the returned instance of YModule is not shown as on-line.

Returns :
an instance of YModule

files→get_module_async() YFiles
files→module_async()

Gets the YModule object for the device on which the function is located (asynchronous version).

```
js  function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as on-line.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking Firefox JavaScript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous JavasSript calls for more details.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

files→get_serialNumber()
files→serialNumber()

YFiles

Returns the serial number of the module, as set by the factory.

```
js      function get_serialNumber( )
cpp     string get_serialNumber( )
m       -(NSString*) serialNumber
pas     string get_serialNumber( ): string
vb      function get_serialNumber( ) As String
cs      string get_serialNumber( )
java    String get_serialNumber( )
uwp     async Task<string> get_serialNumber( )
py      get_serialNumber( )
php     function get_serialNumber( )
ts      async get_serialNumber( ): Promise<string>
es      async get_serialNumber( )
dnp     string get_serialNumber( )
cp      string get_serialNumber( )
cmd     YFiles target get_serialNumber
```

Returns :

a string corresponding to the serial number of the module, as set by the factory.

On failure, throws an exception or returns YFunction.SERIALNUMBER_INVALID.

files→get_userData()
files→userData()

YFiles

Returns the value of the userData attribute, as previously stored using method `set_userData`.

js	function **get_userData**( )
cpp	void * **get_userData**( )
m	-(id) userData
pas	Tobject **get_userData**( ): Tobject
vb	function **get_userData**( ) As Object
cs	object **get_userData**( )
java	Object **get_userData**( )
py	**get_userData**( )
php	function **get_userData**( )
ts	async **get_userData**( ): Promise<object|null>
es	async **get_userData**( )

This attribute is never touched directly by the API, and is at disposal of the caller to store a context.

Returns :
the object stored previously by the caller.

files→isOnline() YFiles

Checks if the filesystem is currently reachable, without raising any error.

```
js      function isOnline( )
cpp     bool isOnline( )
m       -(BOOL) isOnline
pas     boolean isOnline( ): boolean
vb      function isOnline( ) As Boolean
cs      bool isOnline( )
java    boolean isOnline( )
py      isOnline( )
php     function isOnline( )
ts      async isOnline( ): Promise<boolean>
es      async isOnline( )
dnp     bool isOnline( )
cp      bool isOnline( )
```

If there is a cached value for the filesystem in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the filesystem.

Returns :

`true` if the filesystem can be reached, and `false` otherwise

files→isOnline_async() YFiles

Checks if the filesystem is currently reachable, without raising any error (asynchronous version).

```
js  function isOnline_async( callback, context)
```

If there is a cached value for the filesystem in cache, that has not yet expired, the device is considered reachable. No exception is raised if there is an error while trying to contact the device hosting the requested function.

This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking the Javascript virtual machine.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the boolean result

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

files→isReadOnly() YFiles

Test if the function is readOnly.

```
cpp     bool isReadOnly( )
m       -(bool) isReadOnly
pas     boolean isReadOnly( ): boolean
vb      function isReadOnly( ) As Boolean
cs      bool isReadOnly( )
java    boolean isReadOnly( )
uwp     async Task<bool> isReadOnly( )
py      isReadOnly( )
php     function isReadOnly( )
ts      async isReadOnly( ): Promise<boolean>
es      async isReadOnly( )
dnp     bool isReadOnly( )
cp      bool isReadOnly( )
cmd     YFiles target isReadOnly
```

Return `true` if the function is write protected or that the function is not available.

Returns :
`true` if the function is readOnly or not online.

files→load() YFiles

Preloads the filesystem cache with a specified validity duration.

```
js      function load( msValidity)
cpp     YRETCODE load( int msValidity)
m       -(YRETCODE) load : (u64) msValidity
pas     YRETCODE load( msValidity: u64): YRETCODE
vb      function load( ByVal msValidity As Long) As YRETCODE
cs      YRETCODE load( ulong msValidity)
java    int load( long msValidity)
py      load( msValidity)
php     function load( $msValidity)
ts      async load( msValidity: number): Promise<number>
es      async load( msValidity)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

Parameters :

msValidity an integer corresponding to the validity attributed to the loaded function parameters, in milliseconds

Returns :

YAPI.SUCCESS when the call succeeds.

On failure, throws an exception or returns a negative error code.

files→loadAttribute() YFiles

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

```
js      function loadAttribute( attrName)
cpp     string loadAttribute( string attrName)
m       -(NSString*) loadAttribute : (NSString*) attrName
pas     string loadAttribute( attrName: string): string
vb      function loadAttribute( ByVal attrName As String) As String
cs      string loadAttribute( string attrName)
java    String loadAttribute( String attrName)
uwp     async Task<string> loadAttribute( string attrName)
py      loadAttribute( attrName)
php     function loadAttribute( $attrName)
ts      async loadAttribute( attrName: string): Promise<string>
es      async loadAttribute( attrName)
dnp     string loadAttribute( string attrName)
cp      string loadAttribute( string attrName)
```

Parameters :

attrName the name of the requested attribute

Returns :

a string with the value of the the attribute

On failure, throws an exception or returns an empty string.

files→load_async() YFiles

Preloads the filesystem cache with a specified validity duration (asynchronous version).

```
js  function load_async( msValidity, callback, context)
```

By default, whenever accessing a device, all function attributes are kept in cache for the standard duration (5 ms). This method can be used to temporarily mark the cache as valid for a longer period, in order to reduce network traffic for instance.

This asynchronous version exists only in JavaScript. It uses a callback instead of a return value in order to avoid blocking the JavaScript virtual machine.

Parameters :

msValidity	an integer corresponding to the validity of the loaded function parameters, in milliseconds
callback	callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the error code (or YAPI.SUCCESS)
context	caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

files→muteValueCallbacks() YFiles

Disables the propagation of every new advertised value to the parent hub.

```
js      function muteValueCallbacks( )
cpp     int muteValueCallbacks( )
m       -(int) muteValueCallbacks
pas     LongInt muteValueCallbacks( ): LongInt
vb      function muteValueCallbacks( ) As Integer
cs      int muteValueCallbacks( )
java    int muteValueCallbacks( )
uwp     async Task<int> muteValueCallbacks( )
py      muteValueCallbacks( )
php     function muteValueCallbacks( )
ts      async muteValueCallbacks( ): Promise<number>
es      async muteValueCallbacks( )
dnp     int muteValueCallbacks( )
cp      int muteValueCallbacks( )
cmd     YFiles target muteValueCallbacks
```

You can use this function to save bandwidth and CPU on computers with limited resources, or to prevent unwanted invocations of the HTTP callback. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI.SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

files→nextFiles() YFiles

Continues the enumeration of filesystems started using `yFirstFiles()`.

`js`	function **nextFiles**( )
`cpp`	YFiles * **nextFiles**( )
`m`	-(nullable YFiles*) **nextFiles**
`pas`	TYFiles **nextFiles**( ): TYFiles
`vb`	function **nextFiles**( ) As YFiles
`cs`	YFiles **nextFiles**( )
`java`	YFiles **nextFiles**( )
`uwp`	YFiles **nextFiles**( )
`py`	**nextFiles**( )
`php`	function **nextFiles**( )
`ts`	**nextFiles**( ): YFiles | null
`es`	**nextFiles**( )

Caution: You can't make any assumption about the returned filesystems order. If you want to find a specific a filesystem, use `Files.findFiles()` and a hardwareID or a logical name.

Returns :

a pointer to a `YFiles` object, corresponding to a filesystem currently online, or a `null` pointer if there are no more filesystems to enumerate.

files→registerValueCallback() YFiles

Registers the callback function that is invoked on every change of advertised value.

js	function **registerValueCallback**(**callback**)
cpp	int **registerValueCallback**(YFilesValueCallback **callback**)
m	-(int) **registerValueCallback** : (YFilesValueCallback _Nullable) **callback**
pas	LongInt **registerValueCallback**(**callback**: TYFilesValueCallback): LongInt
vb	function **registerValueCallback**(ByVal **callback** As YFilesValueCallback) As Integer
cs	int **registerValueCallback**(ValueCallback **callback**)
java	int **registerValueCallback**(UpdateCallback **callback**)
uwp	async Task<int> **registerValueCallback**(ValueCallback **callback**)
py	**registerValueCallback**(**callback**)
php	function **registerValueCallback**(**$callback**)
ts	async **registerValueCallback**(**callback**: YFilesValueCallback | null): Promise<number>
es	async **registerValueCallback**(**callback**)

The callback is invoked only during the execution of `ySleep` or `yHandleEvents`. This provides control over the time when the callback is triggered. For good responsiveness, remember to call one of these two functions periodically. To unregister a callback, pass a null pointer as argument.

Parameters :

callback the callback function to call, or a null pointer. The callback function should take two arguments: the function object of which the value has changed, and the character string describing the new advertised value.

files→remove() YFiles

Deletes a file, given by its full path name, from the filesystem.

```
js      function remove( pathname)
cpp     int remove( string pathname)
m       -(int) remove : (NSString*) pathname
pas     LongInt remove( pathname: string): LongInt
vb      function remove( ByVal pathname As String) As Integer
cs      int remove( string pathname)
java    int remove( String pathname)
uwp     async Task<int> remove( string pathname)
py      remove( pathname)
php     function remove( $pathname)
ts      async remove( pathname: string): Promise<number>
es      async remove( pathname)
dnp     int remove( string pathname)
cp      int remove( string pathname)
cmd     YFiles target remove pathname
```

Because of filesystem fragmentation, deleting a file may not always free up the whole space used by the file. However, rewriting a file with the same path name will always reuse any space not freed previously. If you need to ensure that no space is taken by previously deleted files, you can use `format_fs` to fully reinitialize the filesystem.

Parameters :

pathname path and name of the file to remove.

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→set_logicalName()
files→setLogicalName()

YFiles

Changes the logical name of the filesystem.

```
js      function set_logicalName( newval)
cpp     int set_logicalName( string newval)
m       -(int) setLogicalName : (NSString*) newval
pas     integer set_logicalName( newval: string): integer
vb      function set_logicalName( ByVal newval As String) As Integer
cs      int set_logicalName( string newval)
java    int set_logicalName( String newval)
uwp     async Task<int> set_logicalName( string newval)
py      set_logicalName( newval)
php     function set_logicalName( $newval)
ts      async set_logicalName( newval: string): Promise<number>
es      async set_logicalName( newval)
dnp     int set_logicalName( string newval)
cp      int set_logicalName( string newval)
cmd     YFiles target set_logicalName newval
```

You can use `yCheckLogicalName()` prior to this call to make sure that your parameter is valid. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Parameters :

newval a string corresponding to the logical name of the filesystem.

Returns :

`YAPI.SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→set_userData() YFiles
files→setUserData()

Stores a user context provided as argument in the userData attribute of the function.

```
js      function set_userData( data)
cpp     void set_userData( void * data)
m       -(void) setUserData : (id) data
pas     set_userData( data: Tobject)
vb      procedure set_userData( ByVal data As Object)
cs      void set_userData( object data)
java    void set_userData( Object data)
py      set_userData( data)
php     function set_userData( $data)
ts      async set_userData( data: object|null): Promise<void>
es      async set_userData( data)
```

This attribute is never touched by the API, and is at disposal of the caller to store a context.

Parameters :

data any kind of object to be stored

files→unmuteValueCallbacks() YFiles

Re-enables the propagation of every new advertised value to the parent hub.

```
js      function unmuteValueCallbacks( )
cpp     int unmuteValueCallbacks( )
m       -(int) unmuteValueCallbacks
pas     LongInt unmuteValueCallbacks( ): LongInt
vb      function unmuteValueCallbacks( ) As Integer
cs      int unmuteValueCallbacks( )
java    int unmuteValueCallbacks( )
uwp     async Task<int> unmuteValueCallbacks( )
py      unmuteValueCallbacks( )
php     function unmuteValueCallbacks( )
ts      async unmuteValueCallbacks( ): Promise<number>
es      async unmuteValueCallbacks( )
dnp     int unmuteValueCallbacks( )
cp      int unmuteValueCallbacks( )
cmd     YFiles target unmuteValueCallbacks
```

This function reverts the effect of a previous call to `muteValueCallbacks()`. Remember to call the `saveToFlash()` method of the module if the modification must be kept.

Returns :
`YAPI.SUCCESS` when the call succeeds.

On failure, throws an exception or returns a negative error code.

files→upload() YFiles

Uploads a file to the filesystem, to the specified full path name.

```
js      function upload( pathname, content)
cpp     int upload( string pathname, string content)
m       -(int) upload : (NSString*) pathname
                      : (NSData*) content
pas     LongInt upload( pathname: string, content: TByteArray): LongInt
vb      procedure upload( ByVal pathname As String, ByVal content As Byte()
cs      int upload( string pathname, byte[] content)
java    int upload( String pathname, byte[] content)
uwp     async Task<int> upload( string pathname, byte[] content)
py      upload( pathname, content)
php     function upload( $pathname, $content)
ts      async upload( pathname: string, content: Uint8Array): Promise<number>
es      async upload( pathname, content)
dnp     int upload( string pathname, byte[] content)
cp      int upload( string pathname, string content)
cmd     YFiles target upload pathname content
```

If a file already exists with the same path name, its content is overwritten.

Parameters :

pathname path and name of the new file to create

content binary buffer with the content to set

Returns :

YAPI.SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→wait_async() YFiles

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

```
js  function wait_async( callback, context)
ts  wait_async( callback: Function, context: object)
es  wait_async( callback, context)
```

The callback function can therefore freely issue synchronous or asynchronous commands, without risking to block the JavaScript VM.

Parameters :

callback callback function that is invoked when all pending commands on the module are completed. The callback function receives two arguments: the caller-specific context object and the receiving function object.

context caller-specific object that is passed as-is to the callback function

Returns :

nothing.

25. Troubleshooting

25.1. Where to start?

If it is the first time that you use a Yoctopuce module and you do not really know where to start, have a look at the Yoctopuce blog. There is a section dedicated to beginners [1].

25.2. Programming examples don't seem to work

Most of Yoctopuce API programming examples are command line programs and require some parameters to work properly. You have to start them from your operating system command prompt, or configure your IDE to run them with the proper parameters. [2].

25.3. Linux and USB

To work correctly under Linux, the the library needs to have write access to all the Yoctopuce USB peripherals. However, by default under Linux, USB privileges of the non-root users are limited to read access. To avoid having to run the *VirtualHub* as root, you need to create a new *udev* rule to authorize one or several users to have write access to the Yoctopuce peripherals.

To add a new *udev* rule to your installation, you must add a file with a name following the "`##-arbitraryName.rules`" format, in the "`/etc/udev/rules.d`" directory. When the system is starting, *udev* reads all the files with a "`.rules`" extension in this directory, respecting the alphabetical order (for example, the "`51-custom.rules`" file is interpreted AFTER the "`50-udev-default.rules`" file).

The "`50-udev-default`" file contains the system default *udev* rules. To modify the default behavior, you therefore need to create a file with a name that starts with a number larger than 50, that will override the system default rules. Note that to add a rule, you need a root access on the system.

In the `udev_conf` directory of the *VirtualHub* for Linux[3] archive, there are two rule examples which you can use as a basis.

[1] see: http://www.yoctopuce.com/EN/blog_by_categories/for-the-beginners
[2] see: http://www.yoctopuce.com/EN/article/about-programming-examples
[3] http://www.yoctopuce.com/FR/virtualhub.php

Example 1: 51-yoctopuce.rules

This rule provides all the users with read and write access to the Yoctopuce USB peripherals. Access rights for all other peripherals are not modified. If this scenario suits you, you only need to copy the "51-yoctopuce_all.rules" file into the "/etc/udev/rules.d" directory and to restart your system.

```
# udev rules to allow write access to all users
# for Yoctopuce USB devices
SUBSYSTEM=="usb", ATTR{idVendor}=="24e0", MODE="0666"
```

Example 2: 51-yoctopuce_group.rules

This rule authorizes the "yoctogroup" group to have read and write access to Yoctopuce USB peripherals. Access rights for all other peripherals are not modified. If this scenario suits you, you only need to copy the "51-yoctopuce_group.rules" file into the "/etc/udev/rules.d" directory and restart your system.

```
# udev rules to allow write access to all users of "yoctogroup"
# for Yoctopuce USB devices
SUBSYSTEM=="usb", ATTR{idVendor}=="24e0", MODE="0664", GROUP="yoctogroup"
```

25.4. ARM Platforms: HF and EL

There are two main flavors of executable on ARM: HF (Hard Float) binaries, and EL (EABI Little Endian) binaries. These two families are not compatible at all. The compatibility of a given ARM platform with of one of these two families depends on the hardware and on the OS build. ArmHL and ArmEL compatibility problems are quite difficult to detect. Most of the time, the OS itself is unable to make a difference between an HF and an EL executable and will return meaningless messages when you try to use the wrong type of binary.

All pre-compiled Yoctopuce binaries are provided in both formats, as two separate ArmHF et ArmEL executables. If you do not know what family your ARM platform belongs to, just try one executable from each family.

25.5. Powered module but invisible for the OS

If your Yocto-MaxiDisplay-G is connected by USB, if its blue led is on, but if the operating system cannot see the module, check that you are using a true USB cable with data wires, and not a charging cable. Charging cables have only power wires.

25.6. Another process named xxx is already using yAPI

If when initializing the Yoctopuce API, you obtain the "*Another process named xxx is already using yAPI*" error message, it means that another application is already using Yoctopuce USB modules. On a single machine only one process can access Yoctopuce modules by USB at a time. You can easily work around this limitation by using a VirtualHub and the network mode [4].

25.7. Disconnections, erratic behavior

If you Yocto-MaxiDisplay-G behaves erratically and/or disconnects itself from the USB bus without apparent reason, check that it is correctly powered. Avoid cables with a length above 2 meters. If needed, insert a powered USB hub [5] [6].

[4] see: http://www.yoctopuce.com/EN/article/error-message-another-process-is-already-using-yapi
[5] see: http://www.yoctopuce.com/EN/article/usb-cables-size-matters
[6] see: http://www.yoctopuce.com/EN/article/how-many-usb-devices-can-you-connect

25.8. Registering a VirtualHub disconnect an other one

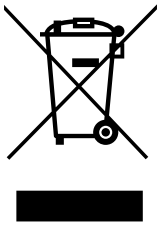
If, when performing a call to RegisterHub() with an VirtualHub address, an other previously registered VirtualHub disconnects, make sure the machine running theses VirtualHubs don't have the same *Hostname*. Same *Hostname* can happen very easily when the operating system is installed from a monolithic image, Raspberry-PI are the best example. The Yoctopuce API uses serial numbers to communicate with devices and VirtualHub serial number are created on the fly based the *hostname* of the machine running the VirtualHub.

25.9. Dropped commands

If, after sending a bunch of commands to a Yoctopuce device, you are under the impression that the last ones have been ignored, typical example is a quick and dirty program meant to configure a device, make sure you used a YAPI.FreeAPI() at the end of the program. Commands are sent to Yoctopuce modules asynchronously thanks to a background thread. When the main program terminates, that thread is killed no matter if some command are left to be sent. However API.FreeAPI () will wait until there is no more commands to send before freeing the API resources and returning.

25.10. Damaged device

Yoctopuce strives to reduce the production of electronic waste. If you believe that your Yocto-MaxiDisplay-G 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-MaxiDisplay-G, do not throw it away in a trash bin but bring it to your local WEEE recycling point. In this way, it will be disposed properly by a specialized WEEE recycling center.

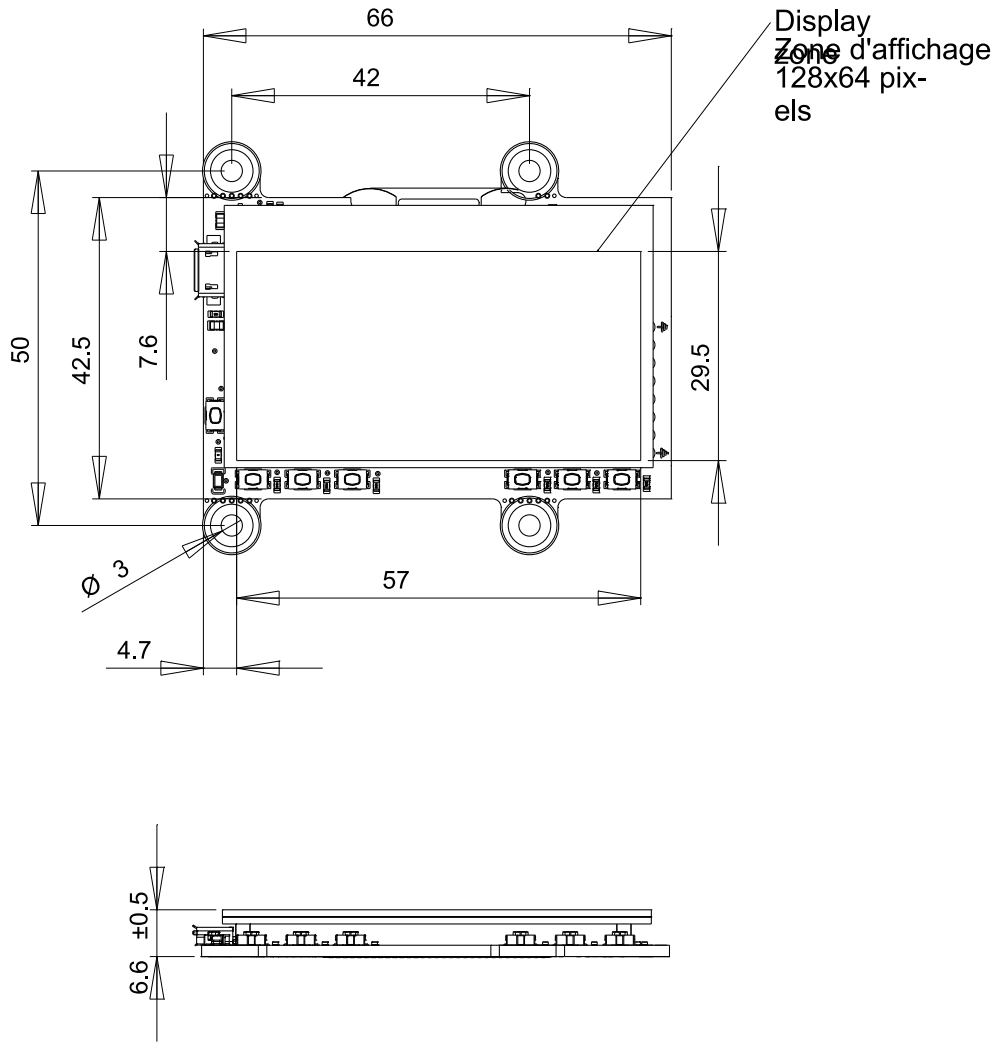
26. Characteristics

You can find below a summary of the main technical characteristics of your Yocto-MaxiDisplay-G module.

Product ID	YD128G64
Hardware release†	
Display area	57x29.5 mm
USB connector	micro-B
Width	58 mm
Length	66 mm
Weight	23 g
Resolution	128x64 px
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	0x004A
Suggested enclosure	YoctoBox-MaxiDisplay
Harmonized tariff code	9032.9000
Made in	Switzerland

† These specifications are for the current hardware revision. Specifications for earlier revisions may differ.

‡ The extended temperature range is defined based on components specifications and has been tested during a limited duration (1h). When using the device in harsh environments for a long period of time, we strongly advise to run extensive tests before going to production.



All dimensions are in mm
Toutes les dimensions sont en mm

Yocto-MaxiDisplay

A4

Scale
1:1
Echelle