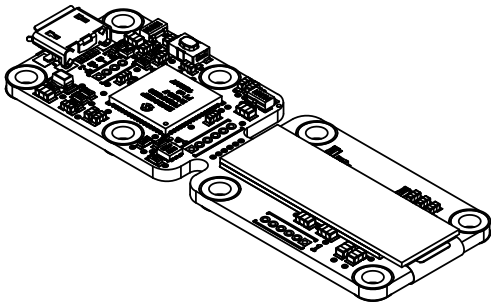
Yocto-MiniDisplay, user manual

Table of contents

1. Introduction

The Yocto-MiniDisplay is a 60x20mm module allowing you to drive a 96 x 16px monochrome OLED screen. 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-MiniDisplay module

Yoctopuce thanks you for buying this Yocto-MiniDisplay 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-MiniDisplay is easy to install anywhere and easy to drive from a maximum of programming languages. If you are nevertheless disappointed with this module, do not hesitate to contact Yoctopuce support[1].

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.

1.1. Prerequisites

In order to use your Yocto-MiniDisplay 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, Mac OS X, Linux, and Android. Yoctopuce modules do not require installing any specific system driver, as they leverage the standard HID driver[2] provided with every operating system.

[1] support@yoctopuce.com

Windows versions currently supported are: Windows XP, Windows 2003, Windows Vista, and Windows 7. Both 32 bit and 64 bit versions are supported. Yoctopuce is frequently testing its modules on Windows XP and Windows 7.

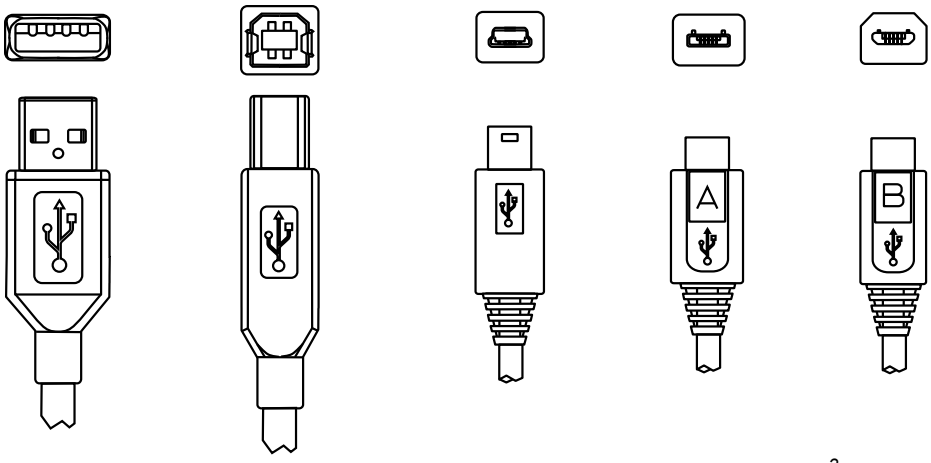
Mac OS X versions currently supported are: 10.6 (Snow Leopard), Mac OS X 10.7 (Lion), and 10.8 (Mountain Lion). Yoctopuce is frequently testing its modules on Mac OS X 10.6 and 10.7.

Linux kernels currently supported are the 2.6 branch and the 3.0 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 2.6.

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 4.x on a Nexus 7 and a Samsung Galaxy S3 with the Java for Android library.

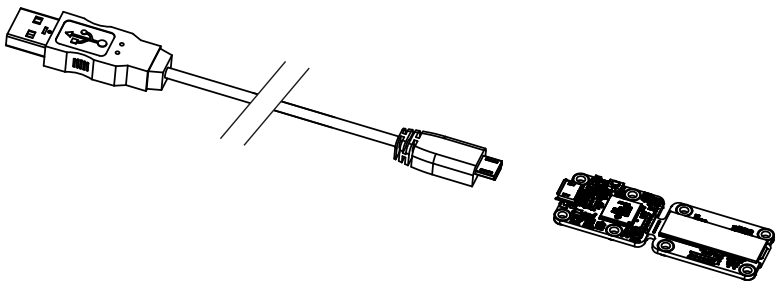
A USB cable, type A-micro B

USB 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 connectors: A, B, Mini B, Micro A, Micro B.[3]

To connect your Yocto-MiniDisplay module to a computer, you need a USB 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 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-MiniDisplay module with a USB cable of type A - micro B.

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

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

1.2. Optional accessories

The accessories below are not necessary to use the Yocto-MiniDisplay module but might be useful depending on your project. These are mostly common products that you can buy from your favourite 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-MiniDisplay module, you can put small screws in the 2.5mm assembly holes, with a screw head no larger than 4.5mm. The best way is to use threaded spacers, which you can then mount wherever you want. You can find more details on this topic in the chapter about assembly and connections.

Micro-USB hub

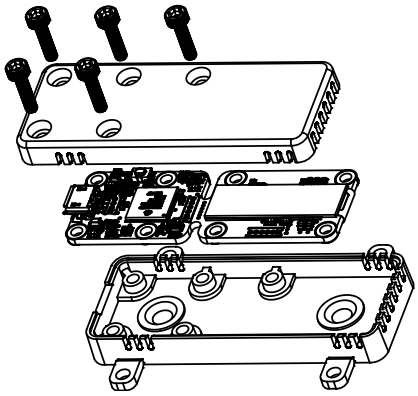
If you intend to put several Yoctopuce modules in a very small space, you can connect them directly to a micro-USB hub. Yoctopuce builds a USB hub particularly small for this purpose (down to 20mmx36mm), on which you can directly solder a USB cable instead of using a USB plug. For more details, see the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

YoctoHub-Ethernet and YoctoHub-Wireless

You can add network connectivity to your Yocto-MiniDisplay, thanks to the YoctoHub-Ethernet and the YoctoHub-Wireless. The YoctoHub-Ethernet provides Ethernet connectivity and the YoctoHub-Wireless provides WiFi connectivity. Both can drive up to three devices and behave exactly like a regular computer running a *VirtualHub*.

Enclosures

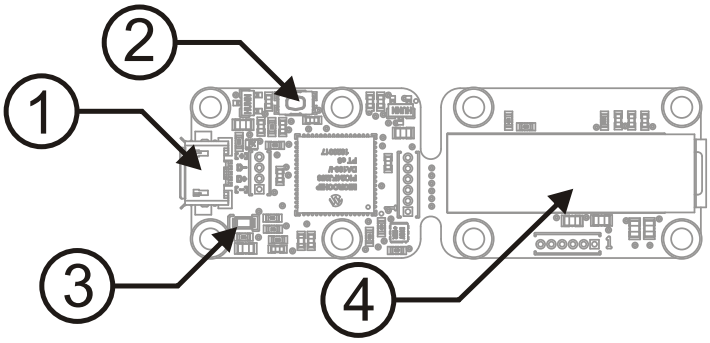
Your Yocto-MiniDisplay has been designed to be installed as is in your project. Nevertheless, Yoctopuce sells enclosures specifically designed for Yoctopuce devices. These enclosures have removable mounting brackets and magnets allowing them to stick on ferromagnetic surfaces. More details are available on the Yoctopuce web site [4]. The suggested enclosure model for your Yocto-MiniDisplay is the YoctoBox-Long-Thin-Black.



You can install your Yocto-MiniDisplay in an optional enclosure

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

2. Presentation



1: Micro-B USB socket 3: Yocto-led
2: Yocto-button 4: OLED dipslay

2.1. Common elements

All Yocto-modules share a number of common functionalities.

USB connector

Yoctopuce modules all come with a micro-B USB socket. The corresponding cables are not the most common, but the sockets are the smallest available.

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.

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-MiniDisplay modules, this number starts with YD096X16. The module can be software driven using this serial number. The serial number cannot be modified.

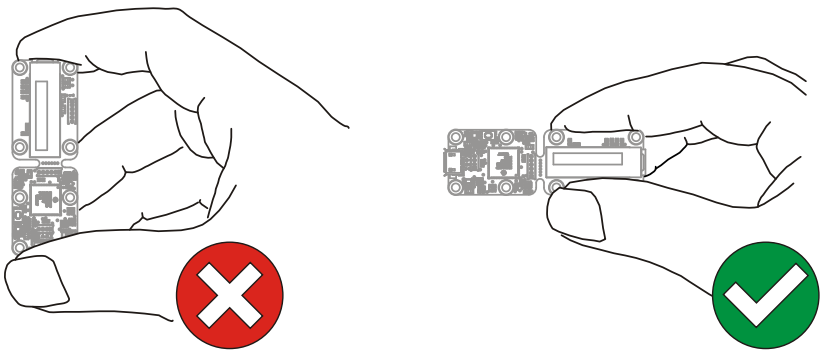
Logical name

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-9616TSWCG02 reference. Being made of glass, it is rather fragile. Do not let your Yocto-MiniDisplay drop. When you install your Yocto-MiniDisplay, 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. Make sure it is not subject to any mechanical constraint. When you manipulate the Yocto-MiniDisplay, do not press on the ribbon.



Do not press on the ribbon cable when you manipulate the Yocto-MiniDisplay.

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

3. Working principles

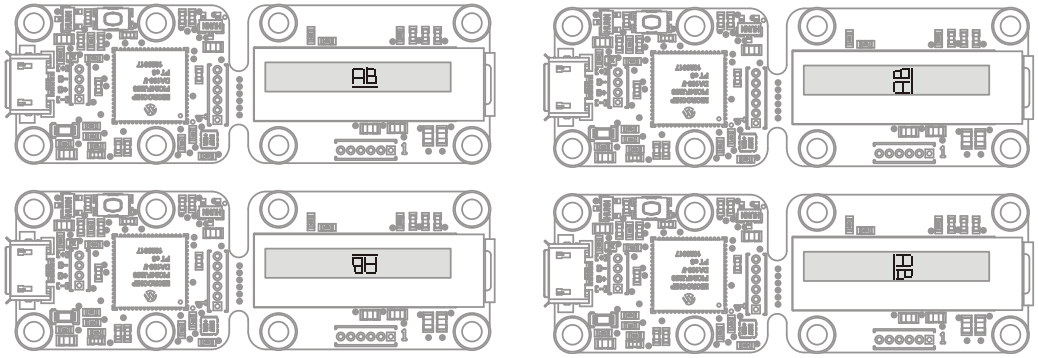
3.1. Embedded processor and memory

Like all the Yoctopuce modules, your Yocto-MiniDisplay 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-MiniDisplay. It does not have to do anything else, everything is managed by the Yocto-MiniDisplay processor. For this reason, the Yocto-MiniDisplay 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-MiniDisplay 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-MiniDisplay 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.

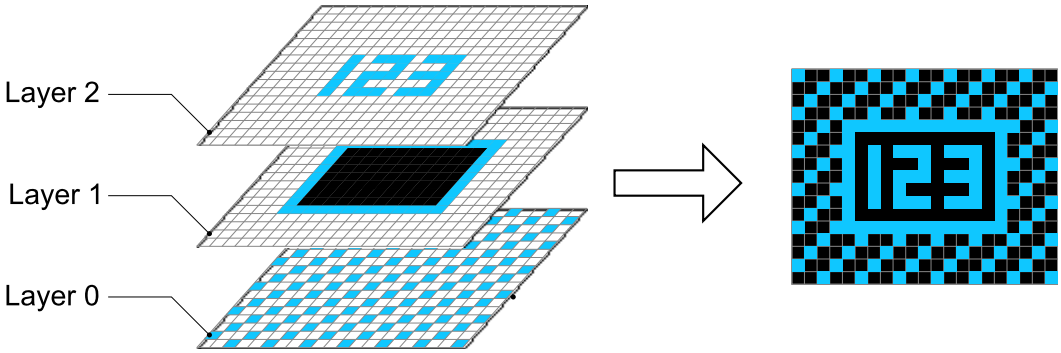


Parameter values LEFT, UP, RIGHT, and DOWN influence the orientation of the display.

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

3.3. Layer system

Your Yocto-MiniDisplay 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-MiniDisplay 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-MiniDisplay 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-MiniDisplay 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-MiniDisplay 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 will

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

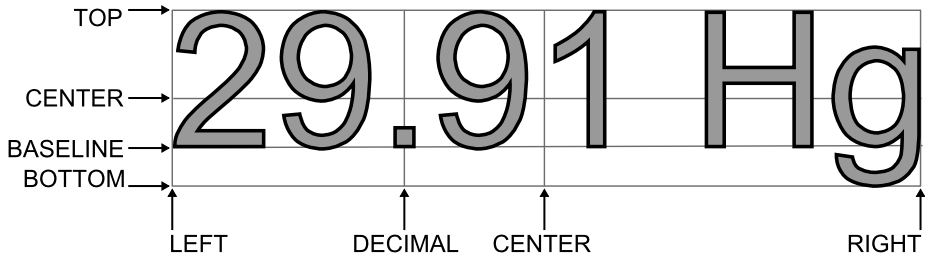
call "white" in this documentation even if it might be light blue for example), black, or transparent (eraser).When you write using the transparent color, the layer below becomes visible. Note that layer 0 is not transparent. Writing in transparent color on this layer is equivalent to writing in black. 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-MiniDisplay 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)
- `8x8.yfm` (monospaced)

Console mode

There is another method to display text on your Yocto-MiniDisplay: 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.

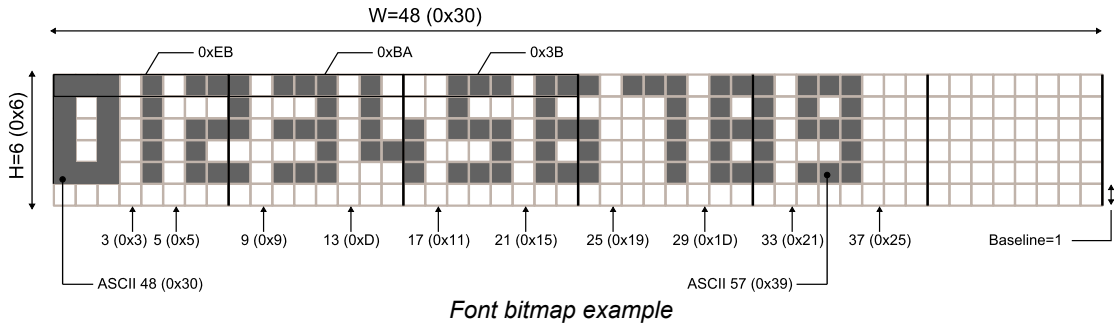
3.6. Font file format

Your Yocto-MiniDisplay 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-MiniDisplay 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
0x07	U8	1	First defined character
0x08	U8	1	Last defined character
0x09	U8	1	Base line (starting from the bottom)
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-MiniDisplay 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-MiniDisplay is powered on, it runs the *yocto.seq* sequence which is hard-coded inside the module. Since the Yocto-MiniDisplay's file system is not persistant you cannot configure it to make it play a custom sequence at startup. But you can disable it: just configure the Yocto-MiniDisplay to play a non existant sequence at startup.

3.8. Optimizations

While the Yocto-MiniDisplay 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-MiniDisplay can be driven without drivers. The transfer 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.

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

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-MiniDisplay 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-MiniDisplay 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

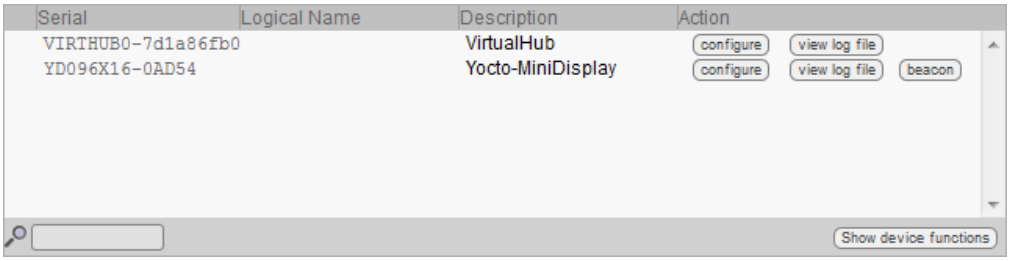
The file system embedded on your Yocto-MiniDisplay has some technical limitations:

- Its maximal storage space is 32KB, allocated in blocks enabling to store up to about 63 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.
- The Yocto-MiniDisplay's filesystem is not persistent: it is implemented in the RAM of the device. It's contents will be wiped out each time the device is powered off.

5. First steps

When reading this chapter, your Yocto-MiniDisplay 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[1]. 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[2]. Start the *Virtual Hub* software in a command line, open your preferred web browser and enter the URL *http://127.0.0.1:4444*. The list of the Yoctopuce modules connected to your computer is displayed.



Module list as displayed in your web bowser.

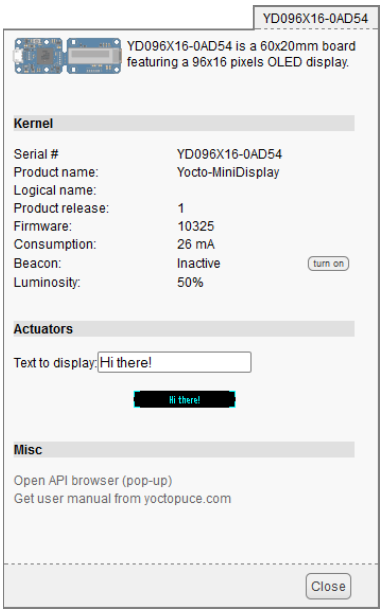
5.1. Localization

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.

5.2. 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-MiniDisplay.

[1] www.yoctopuce.com/EN/virtualhub.php
[2] The interface was tested on FireFox 3+, IE 6+, Safari, and Chrome. It does not work with Opera.

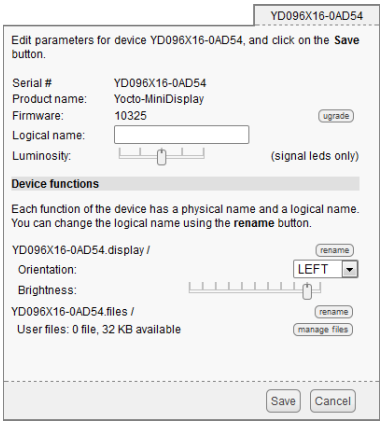


Properties of the Yocto-MiniDisplay module.

This window allows you, among other things, to display an arbitrary text on the screen.

5.3. Configuration

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



Yocto-MiniDisplay module configuration.

Firmware

The module firmware can easily be updated with the help of the interface. To do so, you must beforehand have the adequate firmware on your local disk. 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 [3] in command line [4].

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

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.

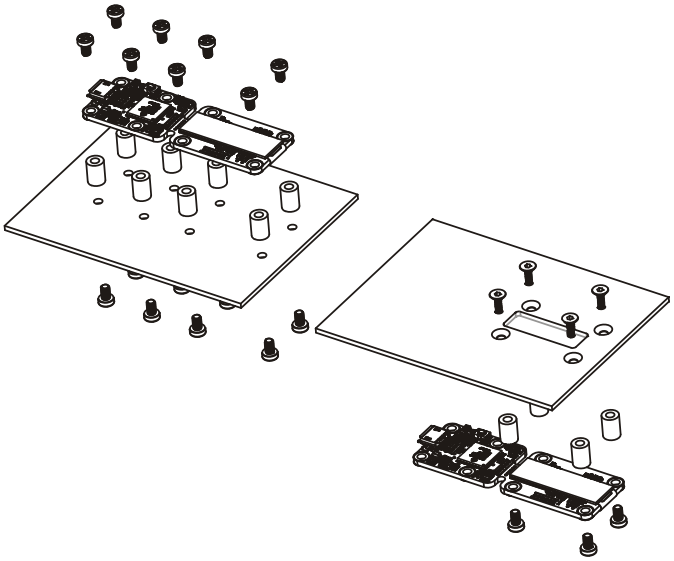
The functions provided by the Yocto-MiniDisplay module are *display*, corresponding to the screen, and *files* corresponding to the file system.

6. Assembly and connections

This chapter provides important information regarding the use of the Yocto-MiniDisplay 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.



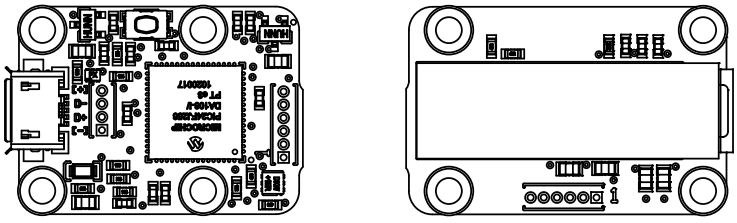
Examples of assembly on supports

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

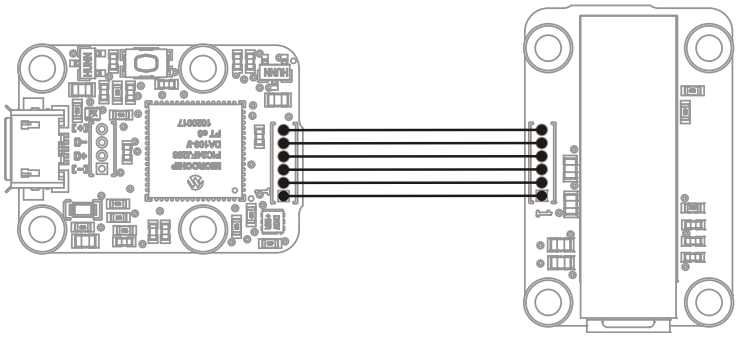
If your intend to screw your module directly against a conducting part, for example a metallic frame, insert an isolating layer in between. Otherwise you are bound to induce a short circuit: there are naked pads under your module. Simple packaging tape should be enough for electric insulation.

6.2. Moving the display away

The Yocto-MiniDisplay module is designed so that you can split it into two parts, allowing you to move away the display from the command sub-module. You can split the module with a hacksaw. When you have split the sub-modules, you can sandpaper the protruding parts without risk.

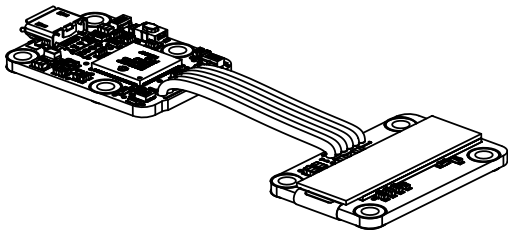


The Yocto-MiniDisplay module is designed so that you can split it into two parts.



Wiring under the modules once separated.

Once the module is split into two, you must rewire the sub-modules. You can connect the sub-modules by soldering simple electric wires, but you can obtain a better result with 1.27 pitch ribbon cable. Consider using solid copper cables, rather than threaded ones: solid copper cables are somewhat less flexible, but much easier to solder. Make sure you don't cross wires, or you will destroy your device.



Moving the display away with a ribbon cable.

6.3. USB power distribution

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

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

In practice, USB would not have been as successful if it was really so picky about power distribution. As it happens, most USB hub manufacturers have been doing savings by not implementing current limitation on ports: they simply connect the computer power supply to every port, and declare themselves as *self-powered hub* even when they are taking all their power from the USB bus (in

order to prevent any power consumption check in the operating system). This looks a bit dirty, but given the fact that computer USB ports are usually well protected by a hardware current limitation around 2000mA, it actually works in every day life, and seldom makes hardware damage.

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

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

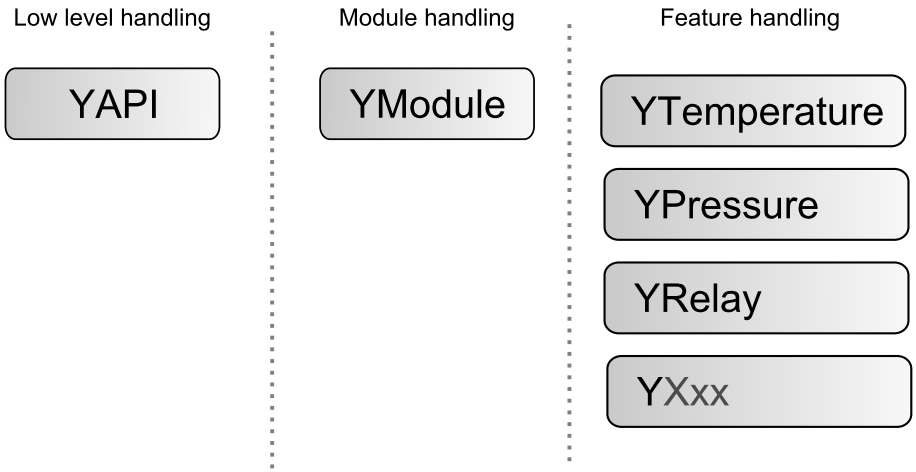
7. Programming, general concepts

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-MiniDisplay 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.

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-MiniDisplay 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-MiniDisplay module

The Yocto-MiniDisplay is a 96x16 OLED display. It includes a RAM-based volatile filesystem to store files (such as images, fonts and animated sequences).

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
usbBandwidth	Enumerated	modifiable

display : Display

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

7.3. Module control interface

This interface is identical for all Yoctopuce USB modules. 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-MiniDisplay module, this serial number always starts with YD096X16. 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 47 at the factory.

productRelease

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

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.

usbBandwidth

Number of USB interfaces used by the device. If this parameter is set to **DOUBLE**, the device can send twice as much data, but this may saturate the USB hub. Remember to call the `saveToFlash()` method and then to reboot the module to apply this setting.

7.4. Display function interface

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).

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 upstraight.

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. Files function interface

The filesystem interface makes it possible to store files on some devices, 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

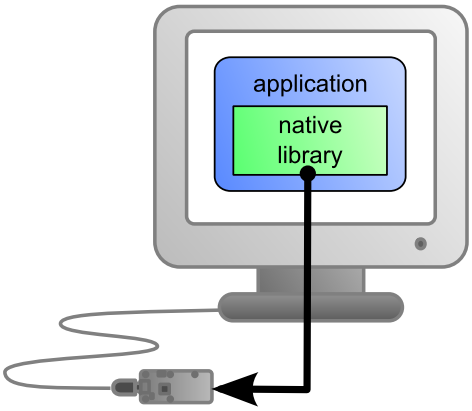
Free space for uploading new files to the filesystem, in bytes.

7.6. What interface: Native, DLL or Service ?

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

Native control

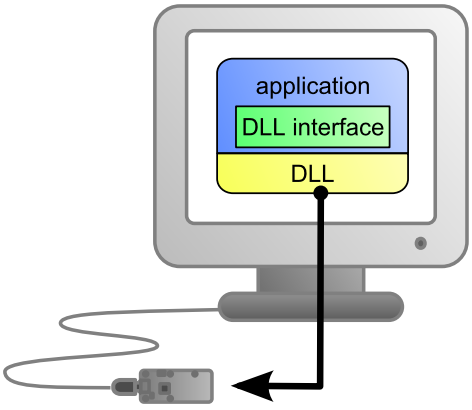
In this case, the software driving your project is compiled directly with a library which provides control of the modules. Objectively, it is the simplest and most elegant solution for the end user. The end user then only needs to plug the USB cable and run your software for everything to work. Unfortunately,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

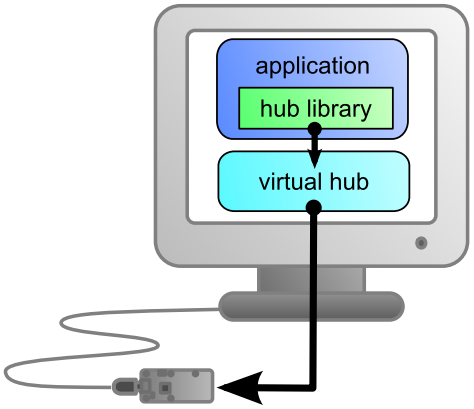
Here, the main part of the code controlling the modules is located in a DLL. The software is compiled with a small library which provides control of the DLL. It is the fastest method to code module support in a given language. Indeed, the "useful" part of the control code is located in the DLL which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

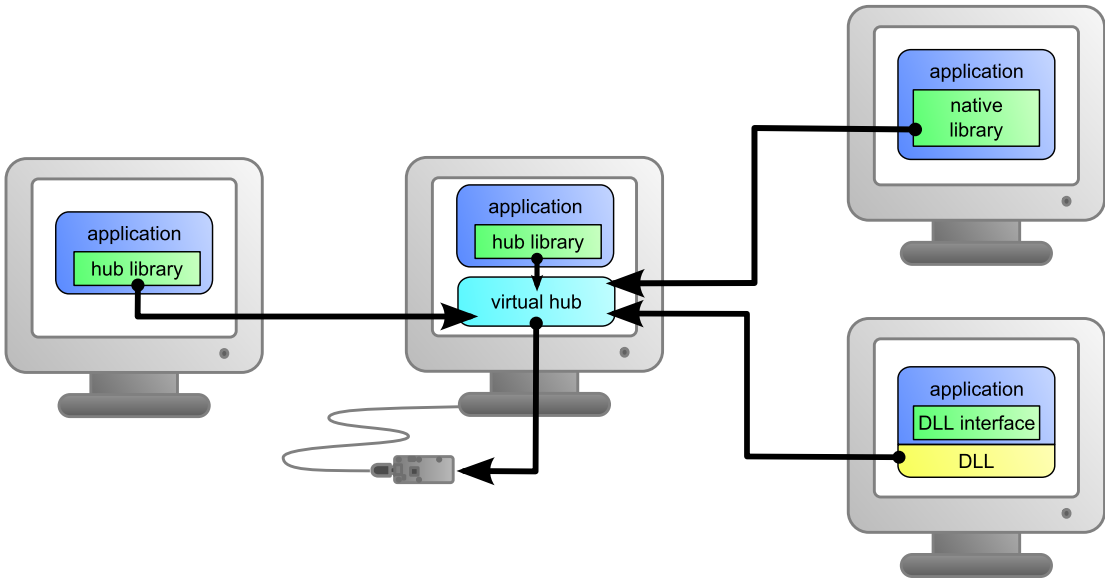
Control by service

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 *Virtual Hub*[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 virtual hub. The end users will have to start the virtual hub before running the project control software itself, unless they decide to install the hub as a service/deamon, in which case the virtual hub starts automatically when the machine starts up.



The application connects itself to the virtual hub 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 virtual hubs.



When a virtual hub 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.

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

Language	Native	Native with DLL	Virtual hub
C++	•	•	•
Objective-C	•	-	•
Delphi	-	•	•
Python	-	•	•
VisualBasic .Net	-	•	•
C# .Net	-	•	•
Javascript	-	-	•
Node.js	-	-	•
PHP	-	-	•
Java	-	-	•
Java for Android	•	-	•
Command line	•	-	•

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.

7.7. Programming, where to start?

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

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

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

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

8. Using the Yocto-MiniDisplay in command line

When you want to perform a punctual operation on your Yocto-MiniDisplay, 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-MiniDisplay, open a shell, and start working by typing for example:

```
C:\>YDisplay any -layer 0 drawText 48 8 CENTER "Hello world!"
```

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

8.2. Use: general description

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

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

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

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

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

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

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

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

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

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

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

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

8.3. Control of the Display function

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

For instance, you can launch:

```
C:\>YDisplay any -layer 0 drawText 48 8 CENTER "Hello world!"
```

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-MiniDisplay module with the *YD096X16-123456* 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 YD096X16-123456.display describe
C:\>YDisplay YD096X16-123456.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 YD096X16-12346 set_logicalName MonPremierModule

C:\>YModule YD096X16-12346 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 YD096X16-12346 set_logicalName MonPremierModule
C:\>YModule YD096X16-12346 saveToFlash
```

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

```
C:\>YModule -s  YD096X16-12346 set_logicalName MonPremierModule
```

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

8.5. Limitations

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

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

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

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

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

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

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

9. Using Yocto-MiniDisplay with Javascript

Javascript is probably not the first language that comes to mind to control hardware, but its ease of use is a great advantage: with Javascript, you only need a text editor and a web browser to realize your first tests.

At the time of writing, the Javascript library functions with any recent browser ... except Opera. It is likely that Opera will end up working with the Yoctopuce library one of these days[1], but it is not the case right now.

Javascript is one of those languages which do not allow you to directly access the hardware layers of your computer. Therefore you need to run the Yoctopuce TCP/IP to USB gateway, named *VirtualHub*, on the machine on which your modules are connected.

9.1. Getting ready

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

- The Javascript 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, connect your modules, run the VirtualHub software, and you are ready to start your first tests. You do not need to install any driver.

9.2. Control of the Display function

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

```
<SCRIPT type="text/javascript" src="yocto_api.js"></SCRIPT>
<SCRIPT type="text/javascript" src="yocto_display.js"></SCRIPT>

// Get access to your device, through the VirtualHub running locally
yRegisterHub('http://127.0.0.1:4444/');
var display = yFindDisplay("YD096X16-123456.display");

// Check that the module is online to handle hot-plug
if(display.isOnline())
```

[1] Actually, as soon as Opera implements support for the HTTP Access-Control-Allow-Origin header.
[2] www.yoctopuce.com/EN/libraries.php
[3] www.yoctopuce.com/EN/virtualhub.php

```
{
    // Use display.get_displayLayer(), ...
}
```

Let us look at these lines in more details.

yocto_api.js and yocto_display.js

These two Javascript includes provide access to functions allowing you to manage Yoctopuce modules. `yocto_api.js` must always be included, `yocto_display.js` is necessary to manage modules containing a display, such as Yocto-MiniDisplay.

yRegisterHub

The `yRegisterHub` 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.

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-MiniDisplay module with serial number *YD096X16-123456* 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.

```
var display = yFindDisplay("YD096X16-123456.display");
var display = yFindDisplay("YD096X16-123456.MyFunction");
var display = yFindDisplay("MyModule.display");
var display = yFindDisplay("MyModule.MyFunction");
var 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

Open your preferred text editor[4], copy the code sample below, save it in the same directory as the Yoctopuce library files and then use your preferred web browser to access this page. The code is also provided in the directory **Examples/Doc-GettingStarted-Yocto-MiniDisplay** 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.

The example is coded to be used either from a web server, or directly by opening the file on the local machine. Note that this latest solution does not work with some versions of Internet Explorer, in particular IE 9 on Windows 7, which is not able to open network connections when working on a local

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

file. In order to use Internet Explorer, you should load the example from a web server. No such problem exists with Chrome, Firefox or Safari.

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

```
<HTML>
<HEAD>
 <TITLE>Hello World</TITLE>
 <SCRIPT type="text/javascript" src="yocto_api.js"></SCRIPT>
 <SCRIPT type="text/javascript" src="yocto_display.js"></SCRIPT>
 <SCRIPT language='javascript1.5' type='text/JavaScript'>
 <!--

 // Setup the API to use the VirtualHub on local machine
 if(yRegisterHub('http://127.0.0.1:4444/') != YAPI_SUCCESS) {
     alert("Cannot contact VirtualHub on 127.0.0.1");
 }

 var disp=null;
 var lastSerial ='';
 var h=0;
 var w=0;
 var l1=null;
 var x,y,vx,vy;

 function run()
 {   var serial = document.getElementById('serial').value;
     if ((disp==null) || (lastSerial != serial))
     { if(serial == '')
       { // Detect any connected module suitable for the demo
         disp = yFirstDisplay();
         if(disp)
           { serial = disp.module().get_serialNumber();
             document.getElementById('serial').value = serial;
           }
       }
       disp = yFindDisplay(serial+".display");
        if(disp.isOnline())
          document.getElementById('msg').value = '';
             else
         { document.getElementById('msg').value = 'Module not connected';
           return;
         }

      // retreive the display size
      w=disp.get_displayWidth();
      h=disp.get_displayHeight();

      // retreive the first layer
      var l0=disp.get_displayLayer(0);

      disp.resetAll();
      l0.clear();

      // display a text in the middle of the screen
      l0.drawText(w / 2, h / 2, Y_ALIGN_CENTER, "Hello world!" );

      // visualize each corner
      l0.moveTo(0,5);l0.lineTo(0,0);l0.lineTo(5,0);
      l0.moveTo(0,h-6);l0.lineTo(0,h-1);l0.lineTo(5,h-1);
      l0.moveTo(w-1,h-6);l0.lineTo(w-1,h-1);l0.lineTo(w-6,h-1);
      l0.moveTo(w-1,5);l0.lineTo(w-1,0);l0.lineTo(w-6,0);

      // draw a circle in the top left corner of layer 1
      l1=disp.get_displayLayer(1);
      l1.clear();
      l1.drawCircle(h / 8, h / 8, h / 8);
        x=0; y=0; vx=1; vy=1;
      setTimeout('refresh()',20);
     }
  }

 function refresh()
 {
```

```
    x+=vx;y+=vy;
    if ((x<0) || (x>w-(h / 4)))  vx=-vx;
    if ((y<0) || (y>h-(h / 4)))  vy=-vy;
    l1.setLayerPosition(x,y,0);
    setTimeout('refresh()',10);
 }


 -->
 </SCRIPT>
</HEAD>
<BODY onload='run();'>
 Module to use: <input id='serial'>
 <input id='msg' style='color:red;border:none;' readonly><br>

</BODY>
</HTML>
```

9.3. 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>
 <SCRIPT type="text/javascript" src="yocto_api.js"></SCRIPT>
 <SCRIPT language='javascript1.5'  type='text/JavaScript'>
 <!--
 // Use explicit error handling rather than exceptions
 yDisableExceptions();

 // Setup the API to use the VirtualHub on local machine
 if(yRegisterHub('http://127.0.0.1:4444/') != YAPI_SUCCESS) {
     alert("Cannot contact VirtualHub on 127.0.0.1");
 }

 var module;

 function refresh()
 {
     var serial = document.getElementById('serial').value;
     if(serial == '') {
         // Detect any conected module suitable for the demo
         module = yFirstModule().nextModule();
         if(module) {
             serial = module.get_serialNumber();
             document.getElementById('serial').value = serial;
         }
     }

     module = yFindModule(serial);
     if(module.isOnline()) {
         document.getElementById('msg').value = '';
         var html = 'serial: '+module.get_serialNumber()+'<br>';
         html += 'logical name: '+module.get_logicalName()+'<br>';
         html += 'luminosity:'+module.get_luminosity()+'%<br>';
         html += 'beacon:';
         if (module.get_beacon()==Y_BEACON_ON)
             html+="ON <a href='javascript:beacon(Y_BEACON_OFF)'>switch off</a><br>";
         else
             html+="OFF <a href='javascript:beacon(Y_BEACON_ON)'>switch on</a><br>";

         html += 'upTime: '+parseInt(module.get_upTime()/1000)+' sec<br>';
         html += 'USB current: '+module.get_usbCurrent()+' mA<br>';
         html += 'logs:<br><pre>'+module.get_lastLogs()+'</pre><br>';
         document.getElementById('data').innerHTML = html;
     } else {
         document.getElementById('msg').value = 'Module not connected';
     }
     setTimeout('refresh()',1000);
 }
```

```
function beacon(state)
{
    module.set_beacon(state);
    refresh();
}
-->
</SCRIPT>
</HEAD>
<BODY onload='refresh();'>
Module to use: <input id='serial'>
<input id='msg' style='color:red;border:none;' readonly><br>
<span id='data'></span>
</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>Change module settings</TITLE>
<SCRIPT type="text/javascript" src="yocto_api.js"></SCRIPT>
<SCRIPT language='javascript1.5'  type='text/JavaScript'>
<!--
// Use explicit error handling rather than exceptions
yDisableExceptions();

// Setup the API to use the VirtualHub on local machine
if(yRegisterHub('http://127.0.0.1:4444/') != YAPI_SUCCESS) {
    alert("Cannot contact VirtualHub on 127.0.0.1");
}

var module;

function refresh()
{
    var serial = document.getElementById('serial').value;
    if(serial == '') {
        // Detect any conected module suitable for the demo
        module = yFirstModule().nextModule();
        if(module) {
            serial = module.get_serialNumber();
            document.getElementById('serial').value = serial;
        }
    }

    module = yFindModule(serial);
    if(module.isOnline()) {
        document.getElementById('msg').value = '';
        document.getElementById('curName').value = module.get_logicalName();
    } else {
        document.getElementById('msg').value = 'Module not connected';
    }
    setTimeout('refresh()',1000);
}

function save()
{
    var newname = document.getElementById('newName').value;
    if (!yCheckLogicalName(newname)) {
        alert('invalid logical name');
        return;
    }
```

```
    module.set_logicalName(newname);
    module.saveToFlash();
 }
 -->
 </SCRIPT>
</HEAD>
<BODY onload='refresh();'>
 Module to use: <input id='serial'>
 <input id='msg' style='color:red;border:none;' readonly><br>
 Current name: <input id='curName' readonly><br>
 New logical name: <input id='newName'>
 <a href='javascript:save();'>Save</a>
</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>Modules inventory</TITLE>
 <SCRIPT type="text/javascript" src="yocto_api.js"></SCRIPT>
 <SCRIPT language='javascript1.5'  type='text/JavaScript'>
 <!--
 // Use explicit error handling rather than exceptions
 yDisableExceptions();

 // Setup the API to use the VirtualHub on local machine
 if(yRegisterHub('http://127.0.0.1:4444/') != YAPI_SUCCESS) {
    alert("Cannot contact VirtualHub on 127.0.0.1");
 }

 function refresh()
 {
    yUpdateDeviceList();

    var htmlcode = '';
    var module = yFirstModule();
    while(module) {
        htmlcode += module.get_serialNumber()
                  +'('+module.get_productName()+")<br>";
        module = module.nextModule();
    }
    document.getElementById('list').innerHTML=htmlcode;
    setTimeout('refresh()',500);
 }
 -->
 </SCRIPT>
</HEAD>
<BODY onload='refresh();'>
 <H1>Device list</H1>
 <tt><span id='list'></span></tt>
</BODY>
</HTML>
```

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

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

10. Using Yocto-MiniDisplay with PHP

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

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

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

10.1. Getting ready

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

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

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

10.2. Control of the Display function

A few lines of code are enough to use a Yocto-MiniDisplay. 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
yRegisterHub('http://127.0.0.1:4444/',$errmsg);
$display = yFindDisplay("YD096X16-123456.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-MiniDisplay.

yRegisterHub

The `yRegisterHub` 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.

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 use logical names as well, as long as you have initialized them. Let us imagine a Yocto-MiniDisplay module with serial number *YD096X16-123456* 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("YD096X16-123456.display");
$display = yFindDisplay("YD096X16-123456.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

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

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

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

  // Use explicit error handling rather than exceptions
  yDisableExceptions();

  // Setup the API to use the VirtualHub on local machine
  if(yRegisterHub('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
      $disp = yFindDisplay("$serial.display");
      if (!$disp->isOnline()) {
          die("Module not connected (check serial and USB cable)");
      }
  } else {
      // or use any connected module suitable for the demo
      $disp = yFirstDisplay();
      if(is_null($disp)) {
          die("No module connected (check USB cable)");
      }
   }
  $serial = $disp->get_module()->get_serialNumber();
  Print("Module to use: <input name='serial' value='$serial'><br>");

  $disp->resetAll();
  // retreive the display size
  $w=$disp->get_displayWidth();
  $h=$disp->get_displayHeight();

  // reteive the first layer
  $l0=$disp->get_displayLayer(0);
  $l0->clear();

  // display a text in the middle of the screen
  $l0->drawText($w / 2, $h / 2, YDisplayLayer::ALIGN_CENTER, "Hello world!" );

  // visualize each corner
  $l0->moveTo(0,5);      $l0->lineTo(0,0);      $l0->lineTo(5,0);
  $l0->moveTo(0,$h-6);   $l0->lineTo(0,$h-1);   $l0->lineTo(5,$h-1);
  $l0->moveTo($w-1,$h-6);$l0->lineTo($w-1,$h-1);$l0->lineTo($w-6,$h-1);
  $l0->moveTo($w-1,5);   $l0->lineTo($w-1,0);   $l0->lineTo($w-6,0);

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

10.3. Control of the module part

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

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

```
  // Use explicit error handling rather than exceptions
  yDisableExceptions();

  // Setup the API to use the VirtualHub on local machine
  if(yRegisterHub('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
      $module = yFindModule("$serial");
      if (!$module->isOnline()) {
          die("Module not connected (check serial and USB cable)");
      }
  } else {
      // or use any connected module suitable for the demo
      $module = yFirstModule();
      if($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());
?>
<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
  yDisableExceptions();

  // Setup the API to use the VirtualHub on local machine
  if(yRegisterHub('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
      $module = yFindModule("$serial");
      if (!$module->isOnline()) {
          die("Module not connected (check serial and USB cable)");
      }
  } else {
      // or use any connected module suitable for the demo
      $module = yFirstModule();
      if($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>");
?>
<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');
    yRegisterHub("http://127.0.0.1:4444/");
    $module   = yFirstModule();
    while (!is_null($module)) {
        printf("%s (%s)<br>", $module->get_serialNumber(),
               $module->get_productName());
```

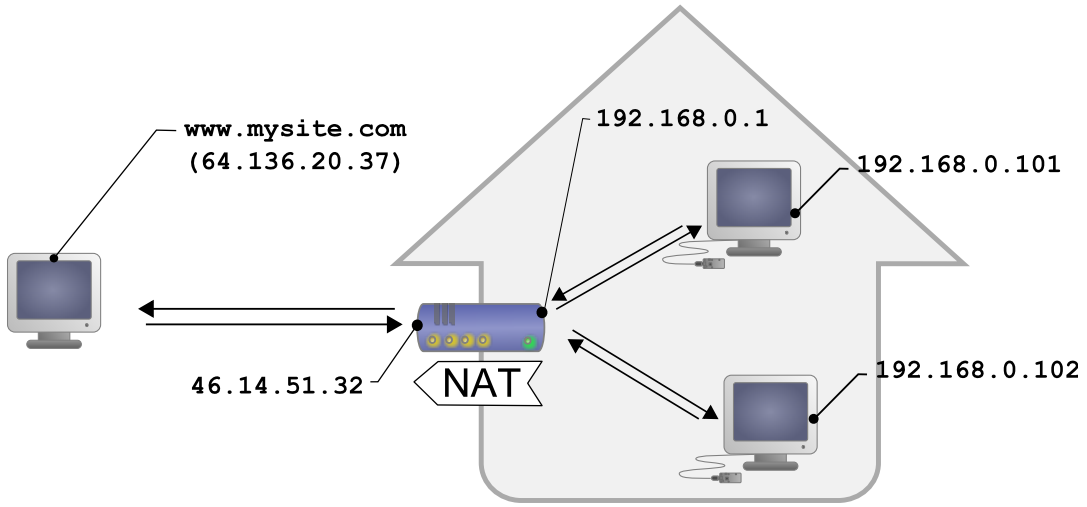
```
        $module=$module->nextModule();
    }
?>
</TT>
</BODY>
</HTML>
```

10.4. HTTP callback API and NAT filters

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

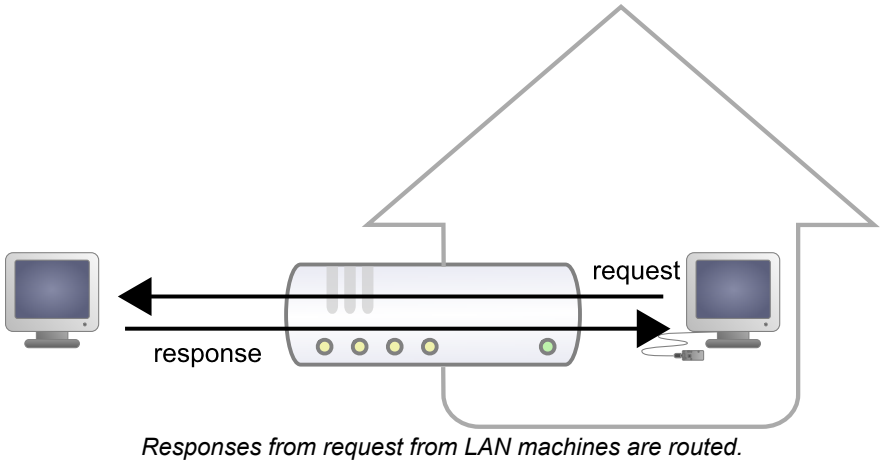
The NAT filter: advantages and disadvantages

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

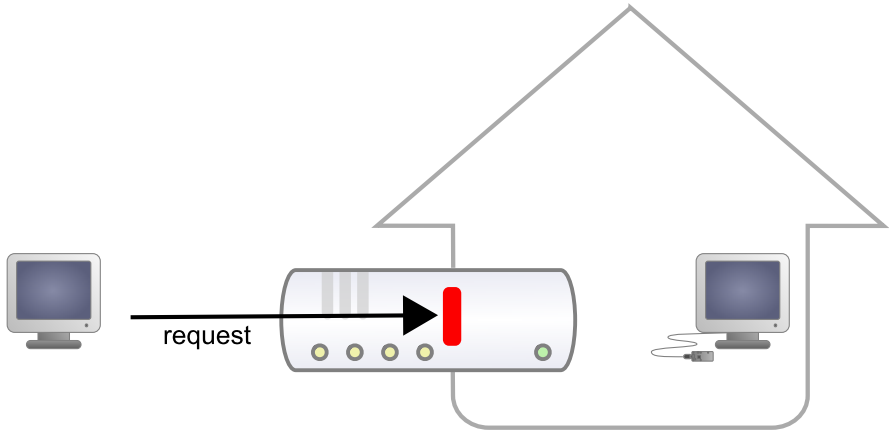


Typical DSL configuration: LAN machines are isolated from the outside by the DSL router

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



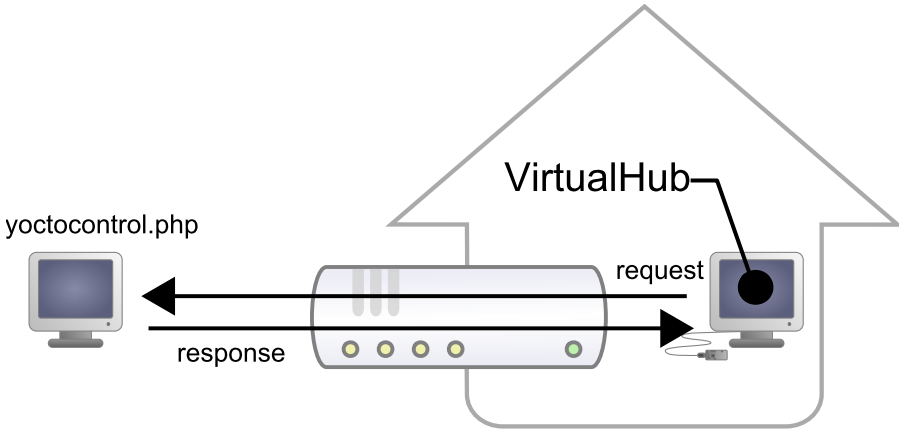
Responses from request from LAN machines are routed.



But requests from the outside are blocked.

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

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



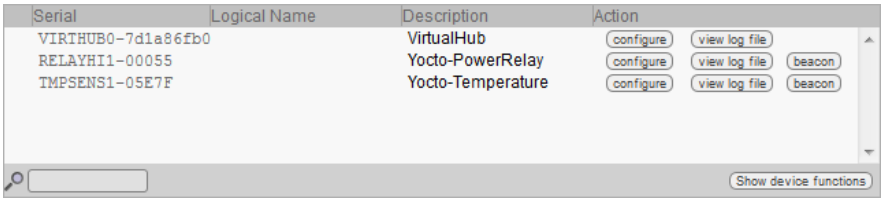
The HTTP callback API uses the VirtualHub which initiates the requests.

Configuration

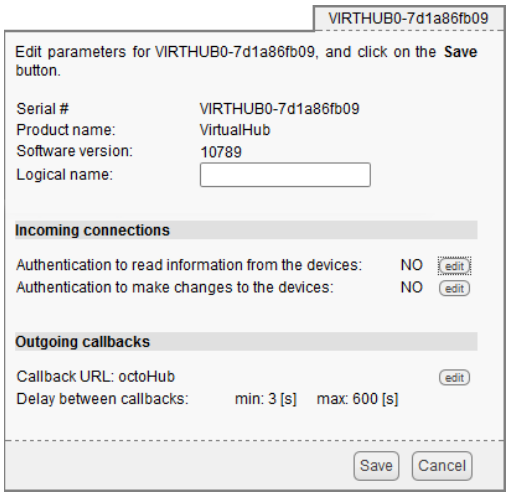
The callback API thus uses the *VirtualHub* as a gateway. All the communications are initiated by the *VirtualHub*. They are thus outgoing communications and therefore perfectly authorized by the DSL router.

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

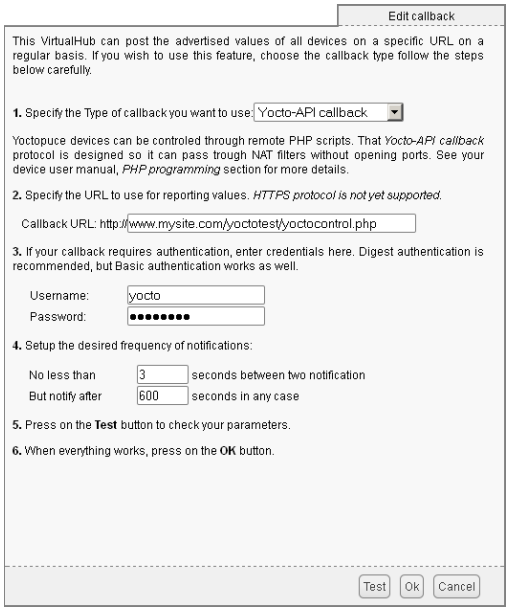
1. Launch a *VirtualHub*
2. Access its interface, usually 127.0.0.1:4444
3. Click on the **configure** button of the line corresponding to the *VirtualHub* itself
4. Click on the **edit** button of the **Outgoing callbacks** section



Click on the "configure" button on the first line



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



And select "Yocto-API callback".

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

Usage

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

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

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

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

Common issues

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

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

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

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

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

Limitations

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

The *HTTP callback Yocto-API* mode is currently available in PHP and Node.JS only.

10.5. Error handling

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

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

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

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

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

bounds for the returned value. In the case of functions which do not normally return information, the return value is `YAPI_SUCCESS` if everything went well, and a different error code in case of failure.

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

11. Using Yocto-MiniDisplay with C++

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

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

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

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

11.1. Control of the Display function

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

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

[...]
String  errmsg;
YDisplay *display;

// Get access to your device, connected locally on USB for instance
yRegisterHub("usb", errmsg);
display = yFindDisplay("YD096X16-123456.display");
```

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

```
// 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-MiniDisplay.

yRegisterHub

The `yRegisterHub` 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.

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 use logical names as well, as long as you have initialized them. Let us imagine a Yocto-MiniDisplay module with serial number *YD096X16-123456* 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 = yFindDisplay("YD096X16-123456.display");
YDisplay *display = yFindDisplay("YD096X16-123456.MyFunction");
YDisplay *display = yFindDisplay("MyModule.display");
YDisplay *display = yFindDisplay("MyModule.MyFunction");
YDisplay *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 C++ environment and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-MiniDisplay** of the Yoctopuce library. If you prefer to work with your favorite text editor, open the file `main.cpp`, and type `make` to build the example when you are done.

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

```
#include "yocto_api.h"
#include "yocto_display.h"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

using namespace std;
```

```
static void usage(void)
{
  cout << "Wrong command line arguments" << endl;
  cout << "usage: demo <serial_number>" << endl;
  cout << "       demo <logical_name>" << endl;
  cout << "       demo any (use any discovered device)" << endl;
  u64 now = yGetTickCount();    // dirty active wait loop
  while (yGetTickCount()-now<3000);
  exit(1);
}

int main(int argc, const char * argv[])
{
  string         errmsg;
  string         target;
  YDisplay      *disp;
  YDisplayLayer *l0,*l1;
  int w,h,x,y,vx,vy;

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

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

  target     = (string) argv[1];
  if (target == "any") {
    disp =  yFirstDisplay();
    if (disp==NULL) {
      cout << "No module connected (check USB cable)" << endl;
      usage();
      return 1;
    }
  } else {
    disp =  yFindDisplay(target + ".display");
  }

  if (!disp->isOnline()) {
    cout << "Module is offline (check USB cable)" << endl;
    usage();
    return 1;
  }

  disp->resetAll();
  // retreive the display size
  w=disp->get_displayWidth();
  h=disp->get_displayHeight();

  // reteive the first layer
  l0=disp->get_displayLayer(0);
  l0->clear();

  // display a text in the middle of the screen
  l0->drawText(w / 2, h / 2, YDisplayLayer::ALIGN_CENTER, "Hello world!" );
  // visualize each corner
  l0->moveTo(0,5);l0->lineTo(0,0);l0->lineTo(5,0);
  l0->moveTo(0,h-6);l0->lineTo(0,h-1);l0->lineTo(5,h-1);
  l0->moveTo(w-1,h-6);l0->lineTo(w-1,h-1);l0->lineTo(w-6,h-1);
  l0->moveTo(w-1,5);l0->lineTo(w-1,0);l0->lineTo(w-6,0);

  // draw a circle in the top left corner of layer 1
  l1=disp->get_displayLayer(1);
  l1->clear();
  l1->drawCircle(h / 8, h / 8, h / 8);

  // and animate the layer
  cout << "Use Ctrl-C to stop";
  x=0; y=0; vx=1; vy=1;
  while (true) {
    x+=vx;y+=vy;
    if ((x<0) || (x>w-(h / 4)))
      vx=-vx;
```

```
    if ((y<0) || (y>h-(h / 4)))
      vy=-vy;
    l1->setLayerPosition(x,y,0);
    YAPI::Sleep(5,errmsg);
  }
  return 0;
}
```

11.2. Control of the module part

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

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

#include "yocto_api.h"

using namespace std;

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


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

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

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

    YModule *module = yFindModule(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;
    }
    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(yRegisterHub("usb", errmsg) != YAPI_SUCCESS) {
        cerr << "RegisterHub error: " << errmsg << endl;
        return 1;
    }

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

    YModule *module = yFindModule(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;
    }
    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(yRegisterHub("usb", errmsg) != YAPI_SUCCESS) {
        cerr << "RegisterHub error: " << errmsg << endl;
        return 1;
    }

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

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

11.3. Error handling

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

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

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

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

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

When you work without exceptions, you can obtain an error code and an error message explaining the source of the error. You can request them from the object which returned the error, calling the

`errType()` and `errMessage()` methods. Their returned values contain the same information as in the exceptions when they are active.

11.4. Integration variants for the C++ Yoctopuce library

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

Integration in source format

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

Integration of the Yoctopuce library as a static library is a simpler manner to build a small executable which uses Yoctopuce modules. You can quickly compile the program with a single command. You do not need to install a dynamic library specific to Yoctopuce, everything is in the executable.

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**.

Then, 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 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**.

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

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

With GCC, the command line to compile is simply:

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

12. Using Yocto-MiniDisplay with Objective-C

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

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

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

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

12.1. Control of the Display function

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

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



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

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

```
int main(int argc, const char * argv[])
{
    NSError *error;
    YDisplay *disp;
    YDisplayLayer *l0, *l1;
    int h, w, y, x, vx, vy;
    @autoreleasepool {

        // Setup the API to use local USB devices
        if([YAPI RegisterHub:@"usb": &error] != YAPI_SUCCESS) {
            NSLog(@"RegisterHub error: %@", [error localizedDescription]);
            usage();
            return 1;
        }
        if(argc < 2){
            disp = [YDisplay FirstDisplay];
            if(disp == nil){
                NSLog(@"No module connected (check USB cable)");
                usage();
                return 1;
            }
        } else {
            NSString *target = [NSString stringWithUTF8String:argv[1]];

            disp = [YDisplay FindDisplay:target];
            if(![disp isOnline]){
                NSLog(@"Module not connected (check identification and USB cable)");
                usage();
                return 1;
            }
        }
        // clear screen (and all layers)
        [disp resetAll];
        // retreive the display size
        w = [disp get_displayWidth];
        h = [disp get_displayHeight];

        // reteive the first layer
        l0 = [disp get_displayLayer:0];

        // display a text in the middle of the screen
        [l0 drawText:w / 2 :h / 2 :Y_ALIGN_CENTER  :@"Hello world!"];

        // visualize each corner
        [l0 moveTo:0 :5]; [l0 lineTo:0 :0]; [l0 lineTo:5 :0];
        [l0 moveTo:0 :h - 6]; [l0 lineTo:0 :h - 1]; [l0 lineTo:5 :h - 1];
        [l0 moveTo:w - 1 :h - 6]; [l0 lineTo:w - 1 :h - 1]; [l0 lineTo:w - 6 :h - 1];
        [l0 moveTo:w - 1 :5]; [l0 lineTo:w - 1  :0]; [l0 lineTo:w - 6 :0];

        // draw a circle in the top left corner of layer 1
        l1 = [disp get_displayLayer:1];
        [l1 clear];
        [l1 drawCircle:h / 8 :h / 8 :h / 8];

        // and animate the layer
        NSLog(@"Use Ctrl-C to stop");
        x = 0; y = 0; vx = 1; vy = 1;
        while (true) {
            x += vx; y += vy;
            if ((x < 0) || (x > w - (h / 4))) vx = -vx;
            if ((y < 0) || (y > h - (h / 4))) vy = -vy;
            [l1 setLayerPosition:x :y :0];
            [YAPI Sleep:5 :&error];
        }
        [YAPI FreeAPI];
    }

    return 0;
}
```

There are only a few really important lines in this example. We will look at them in details.

yocto_api.h et 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-MiniDisplay.

yRegisterHub

The `yRegisterHub` 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.

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 use logical names as well, as long as you have initialized them. Let us imagine a Yocto-MiniDisplay module with serial number *YD096X16-123456* 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 = yFindDisplay(@"YD096X16-123456.display");
YDisplay *display = yFindDisplay(@"YD096X16-123456.MyFunction");
YDisplay *display = yFindDisplay(@"MyModule.display");
YDisplay *display = yFindDisplay(@"MyModule.MyFunction");
YDisplay *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.

12.2. Control of the module part

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

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

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


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

    @autoreleasepool {
        // Setup the API to use local USB devices
        if([YAPI RegisterHub:@"usb": &error] != YAPI_SUCCESS) {
            NSLog(@"RegisterHub error: %@", [error localizedDescription]);
            return 1;
        }
        if(argc < 2)
            usage(argv[0]);
```

```
        NSString *serial_or_name =[NSString stringWithUTF8String:argv[1]];
        YModule *module = [YModule FindModule:serial_or_name];  // use serial or logical
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:       %d 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
);
        }
    }
    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(yRegisterHub(@"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]];
        YModule *module = yFindModule(serial_or_name);  // use serial or logical name

        if (module.isOnline) {
            if (argc >= 3){
                NSString *newname =  [NSString stringWithUTF8String:argv[2]];
                if (!yCheckLogicalName(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
);
        }
    }
    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(yRegisterHub(@"usb", &error) != YAPI_SUCCESS) {
            NSLog(@"RegisterHub error: %@\n", [error localizedDescription]);
            return 1;
        }

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

        YModule *module = yFirstModule();
        while (module != nil) {
            NSLog(@"%@ %@",module.serialNumber, module.productName);
            module = [module nextModule];
        }
    }
    return 0;
}
```

12.3. Error handling

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

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

could crash the software. The only way to prevent this is to implement one of the two error handling techniques described below.

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

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

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

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

13. Using Yocto-MiniDisplay with Visual Basic .NET

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

13.1. Installation

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

13.2. Using the Yoctopuce API in a Visual Basic project

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

Configuring a Visual Basic project

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

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

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

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

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

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

13.3. Control of the Display function

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

```
[...]
Dim errmsg As String errmsg
Dim display As YDisplay

REM Get access to your device, connected locally on USB for instance
yRegisterHub("usb", errmsg)
display = yFindDisplay("YD096X16-123456.display")

REM Hot-plug is easy: just check that the device is online
If (display.isOnline()) Then
   REM Use display.get_displayLayer(), ...
End If
```

Let's look at these lines in more details.

yRegisterHub

The yRegisterHub 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.

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 use logical names as well, as long as you have initialized them. Let us imagine a Yocto-MiniDisplay module with serial number *YD096X16-123456* 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("YD096X16-123456.display")
display = yFindDisplay("YD096X16-123456.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.

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

A real example

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

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

```
Module Module1

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

    Sub Main()
        Dim argv() As String = System.Environment.GetCommandLineArgs()
        Dim errmsg As String = ""
        Dim target As String
    Dim disp As YDisplay
    Dim l0, l1 As YDisplayLayer
    Dim h, w, y, x, vx, vy As Integer


    If argv.Length <= 1 Then Usage()

    target = argv(1)



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

        If target = "any" Then
      disp = yFirstDisplay()
      If disp Is Nothing Then
        Console.WriteLine("No module connected (check USB cable) ")
        End
      End If

        Else
      disp = yFindDisplay(target + ".display")

    End If

    If Not (disp.isOnline()) Then
      Console.WriteLine("Module not connected (check identification and USB cable)")
      End
    End If

    REM Display clean up
    disp.resetAll()
    REM retreive the display size
    w = disp.get_displayWidth()
    h = disp.get_displayHeight()

    REM reteive the first layer
    l0 = disp.get_displayLayer(0)

    REM display a text in the middle of the screen
    l0.drawText(CInt(w / 2), CInt(h / 2), Y_ALIGN.CENTER, "Hello world!")

    REM visualize each corner
    l0.moveTo(0, 5)
    l0.lineTo(0, 0)
    l0.lineTo(5, 0)
    l0.moveTo(0, h - 6)
```

```
    l0.lineTo(0, h - 1)
    l0.lineTo(5, h - 1)
    l0.moveTo(w - 1, h - 6)
    l0.lineTo(w - 1, h - 1)
    l0.lineTo(w - 6, h - 1)
    l0.moveTo(w - 1, 5)
    l0.lineTo(w - 1, 0)
    l0.lineTo(w - 6, 0)

    REM draw a circle in the top left corner of layer 1
    l1 = disp.get_displayLayer(1)
    l1.clear()
    l1.drawCircle(CInt(h / 8), CInt(h / 8), CInt(h / 8))

    REM and animate the layer
    Console.WriteLine("Use Ctrl-C to stop")
    x = 0
    y = 0
    vx = 1
    vy = 1

    While (True)
      x += vx
      y += vy
      If ((x < 0) Or (x > w - (h / 4))) Then vx = -vx
      If ((y < 0) Or (y > h - (h / 4))) Then vy = -vy
      l1.setLayerPosition(x, y, 0)
      YAPI.Sleep(5, errmsg)
    End While

  End Sub

End Module
```

13.4. Control of the module part

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

```
Imports System.IO
Imports System.Environment

Module Module1

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

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

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

    If argv.Length < 2 Then usage()

    m = yFindModule(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



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

    m = yFindModule(argv(1)) REM use serial or logical name
    If m.isOnline() Then

      newname = argv(2)
      If (Not yCheckLogicalName(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

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

    Console.WriteLine("Device list")
    M = yFirstModule()
    While M IsNot Nothing
      Console.WriteLine(M.get_serialNumber() + " (" + M.get_productName() + ")")
      M = M.nextModule()
    End While

  End Sub

    End Module
```

13.5. Error handling

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

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

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

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

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

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

14. Using Yocto-MiniDisplay 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].

14.1. Installation

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

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

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

Configuring a Visual C# project

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

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

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

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

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

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

14.3. Control of the Display function

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

```
[...]
string errmsg ="";
YDisplay display;

// Get access to your device, connected locally on USB for instance
YAPI.RegisterHub("usb", errmsg);
display = YDisplay.FindDisplay("YD096X16-123456.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-MiniDisplay module with serial number *YD096X16-123456* 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("YD096X16-123456.display");
display = YDisplay.FindDisplay("YD096X16-123456.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.

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

A real example

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

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

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

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

    static void Main(string[] args)
    {
      string errmsg = "";
      string target;
      YDisplay disp;
      YDisplayLayer l0, l1;
      int h, w, y, x, vx, vy;
      if (args.Length < 1) usage();

      target = args[0].ToUpper();

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

      // find the display according to command line parameters
      if (target == "ANY")
      {
        disp = YDisplay.FirstDisplay();
        if (disp == null)
        {
          Console.WriteLine("No module connected (check USB cable) ");
          Environment.Exit(0);
        }
      }
      else disp = YDisplay.FindDisplay(target + ".display");


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

      //clean up
      disp.resetAll();

      // retreive the display size
      w = disp.get_displayWidth();
      h = disp.get_displayHeight();

      // reteive the first layer
      l0 = disp.get_displayLayer(0);

      // display a text in the middle of the screen
      l0.drawText(w / 2, h / 2, YDisplayLayer.ALIGN.CENTER, "Hello world!");
```

```
      // visualize each corner
      l0.moveTo(0, 5); l0.lineTo(0, 0); l0.lineTo(5, 0);
      l0.moveTo(0, h - 6); l0.lineTo(0, h - 1); l0.lineTo(5, h - 1);
      l0.moveTo(w - 1, h - 6); l0.lineTo(w - 1, h - 1); l0.lineTo(w - 6, h - 1);
      l0.moveTo(w - 1, 5); l0.lineTo(w - 1, 0); l0.lineTo(w - 6, 0);

      // draw a circle in the top left corner of layer 1
      l1 = disp.get_displayLayer(1);
      l1.clear();
      l1.drawCircle(h / 8, h / 8, h / 8);

      // and animate the layer
      Console.WriteLine("Use Ctrl-C to stop");
      x = 0; y = 0; vx = 1; vy = 1;
      while (true)
      {
        x += vx; y += vy;
        if ((x < 0) || (x > w - (h / 4))) vx = -vx;
        if ((y < 0) || (y > h - (h / 4))) vy = -vy;
        l1.setLayerPosition(x, y, 0);
        YAPI.Sleep(5, ref errmsg);
      }
    }
  }
}
```

14.4. Control of the module part

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

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


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

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

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


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

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

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

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

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

   }
  }
}
```

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");
    }
  }
}
```

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();
      }

    }
  }
}
```

14.5. Error handling

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

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

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

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

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

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

15. Using Yocto-MiniDisplay 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.

15.1. 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.

15.2. Control of the Display function

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:

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

[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.

```
Procedure  Usage();
  var
   exe : string;

  begin
    exe:= ExtractFileName(paramstr(0));
    WriteLn(exe+' <serial_number>');
    WriteLn(exe+' <logical_name>');
    WriteLn(exe+' any');
    halt;
  End;

var
  disp      : TYDisplay;
  l0,l1     : TYDisplayLayer;
  errmsg    : string;
  w,h       : integer;
  x,y,vx,vy :integer;


begin


  if (paramcount<1) then usage();

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

  // first one of the two RBG leds
  if paramstr(1)='any' then
    begin
      disp := yFirstDisplay();
      if disp=nil then
         begin
           writeln('No module connected (check USB cable)');
           halt;
         end
      end
   else
  disp:= YFindDisplay(paramstr(1)+'.display');

  // make sure it is online
  if not(disp.isOnline()) then
      begin
        writeln('No module connected (check USB cable)');
        halt;
      end;

  // display clean up
  disp.resetAll();

  // retreive the display size
  w:=disp.get_displayWidth();
  h:=disp.get_displayHeight();

  // reteive the first layer
  L0:=Disp.get_displaylayer(0);

  // display a text in the middle of the screen
  L0.drawText(w div 2, h div 2, Y_ALIGN_CENTER, 'Hello world!' );
  // visualize eah corner
  L0.moveto(0,5);L0.lineto(0,0);L0.lineto(5,0);
  L0.moveto(0,h-6);L0.lineto(0,H-1);L0.lineto(5,H-1);
  L0.moveto(w-1,h-6);L0.lineto(w-1,H-1);L0.lineto(w-6,H-1);
  L0.moveto(w-1,5);L0.lineto(w-1,0);L0.lineto(w-6,0);

  // draw a circle in the top left corner of layer 1
  L1:=Disp.get_displaylayer(1);
  L1.clear();
  L1.drawCircle(H div 8, H div 8, h div 8);

  // and animate the layer
  Writeln('Use Ctrl-C to stop');
```

```
  x:=0; y:=0; vx:=1; vy:=1;
  while (true) do
   begin
    x:=x+vx;y:=y+vy;
    if (x<0) or (x>w-(h div 4)) then vx:=-vx;
    if (y<0) or (y>h-(h div 4)) then vy:=-vy;
    l1.setLayerPosition(x,y,0);
    ysleep(5,errmsg);
   end;

end.
```

There are only a few really important lines in this sample example. We will look at them in 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-MiniDisplay.

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-MiniDisplay module with serial number *YD096X16-123456* 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("YD096X16-123456.display");
display := yFindDisplay("YD096X16-123456.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.

15.3. Control of the module part

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

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

```
const
  serial = 'YD096X16-123456'; // 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';
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 = 'YD096X16-123456'; // 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();

  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;

end.
```

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

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

16. Using the Yocto-MiniDisplay 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].

16.1. 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.

16.2. 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.

16.3. Control of the Display function

A few lines of code are enough to use a Yocto-MiniDisplay. 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

```
[...]

errmsg=YRefParam()
#Get access to your device, connected locally on USB for instance
YAPI.RegisterHub("usb",errmsg)
display = YDisplay.FindDisplay("YD096X16-123456.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-MiniDisplay module with serial number *YD096X16-123456* 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("YD096X16-123456.display")
display = YDisplay.FindDisplay("YD096X16-123456.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-MiniDisplay** of the Yoctopuce library.

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

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

from yocto_api import *
from yocto_display import *

def usage():
    scriptname = os.path.basename(sys.argv[0])
```

```
    print("Usage:")
    print(scriptname+' <serial_number>')
    print(scriptname+' <logical_name>')
    print(scriptname+' any  ')
    sys.exit()

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

errmsg=YRefParam()

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

target=sys.argv[1]

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



if target=='any':
    # retreive any RGB led
    disp = YDisplay.FirstDisplay()
    if disp is None :
        die('No module connected')
else:
    disp= YDisplay.FindDisplay(target + ".display")

if not disp.isOnline():
    die("Module not connected ")

# display clean up
disp.resetAll()

# retreive the display size
w=disp.get_displayWidth()
h=disp.get_displayHeight()

# retreive the first layer
l0=disp.get_displayLayer(0)
l0.clear()

#display a text in the middle of the screen
l0.drawText(w / 2,h / 2, YDisplayLayer.ALIGN.CENTER, "Hello world!" )

# visualize each corner
l0.moveTo(0,5)
l0.lineTo(0,0)
l0.lineTo(5,0)
l0.moveTo(0,h-6)
l0.lineTo(0,h-1)
l0.lineTo(5,h-1)
l0.moveTo(w-1,h-6)
l0.lineTo(w-1,h-1)
l0.lineTo(w-6,h-1)
l0.moveTo(w-1,5)
l0.lineTo(w-1,0)
l0.lineTo(w-6,0)

# draw a circle in the top left corner of layer 1
l1=disp.get_displayLayer(1)
l1.clear()
l1.drawCircle(h / 8, h / 8, h / 8)

# and animate the layer
print("Use Ctrl-C to stop")
x=0
y=0
vx=1
vy=1
while True:
    x+=vx
    y+=vy
    if x<0 or x>w-(h / 4) :  vx=-vx
    if y<0 or y>h-(h / 4)  : vy=-vy
    l1.setLayerPosition(x,y,0)
```

```
    YAPI.Sleep(5,errmsg)
```

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.

```
#!/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)")
```

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")
```

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()
```

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

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

17. Using the Yocto-MiniDisplay 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.

17.1. 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.

17.2. Control of the Display function

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

```
[...]

// Get access to your device, connected locally on USB for instance
YAPI.RegisterHub("127.0.0.1");
display = YDisplay.FindDisplay("YD096X16-123456.display");

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

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

```
[...]
```

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-MiniDisplay module with serial number *YD096X16-123456* 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("YD096X16-123456.display")
display = YDisplay.FindDisplay("YD096X16-123456.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-MiniDisplay** of the Yoctopuce library.

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

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

public class Demo {

    private static void disp(YDisplay display, String text, YDisplayLayer.ALIGN al) throws
YAPI_Exception
    {
        YDisplayLayer layer0 = display.get_displayLayer(0);
        int l = (int) display.get_displayWidth();
        int h = (int) display.get_displayHeight();
        int mx = l / 2;
        int my = h / 2;
        layer0.clear();
        layer0.moveTo(mx, 0);
        layer0.lineTo(mx, h);
        layer0.moveTo(0, my);
        layer0.lineTo(l, my);
        layer0.drawText(mx, my, al, text);
    }
```

```
    public static void main(String[] args)
    {

        YDisplay disp;
        YDisplayLayer l0, l1;
        int h, w, y, x, vx, vy;

        // API init
        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 == 0) {
            disp = YDisplay.FirstDisplay();
            if (disp == null) {
                System.out.println("No module connected (check USB cable)");
                System.exit(1);
            }
        } else {
            disp = YDisplay.FindDisplay(args[0] + ".display");
        }

        try {
            //clean up
            disp.resetAll();

            // retreive the display size
            w = disp.get_displayWidth();
            h = disp.get_displayHeight();

            // reteive the first layer
            l0 = disp.get_displayLayer(0);

            // display a text in the middle of the screen
            l0.drawText(w / 2, h / 2, YDisplayLayer.ALIGN.CENTER, "Hello world!");

            // visualize each corner
            l0.moveTo(0, 5);
            l0.lineTo(0, 0);
            l0.lineTo(5, 0);
            l0.moveTo(0, h - 6);
            l0.lineTo(0, h - 1);
            l0.lineTo(5, h - 1);
            l0.moveTo(w - 1, h - 6);
            l0.lineTo(w - 1, h - 1);
            l0.lineTo(w - 6, h - 1);
            l0.moveTo(w - 1, 5);
            l0.lineTo(w - 1, 0);
            l0.lineTo(w - 6, 0);

            // draw a circle in the top left corner of layer 1
            l1 = disp.get_displayLayer(1);
            l1.clear();
            l1.drawCircle(h / 8, h / 8, h / 8);

            // and animate the layer
            System.out.println("Use Ctrl-C to stop");
            x = 0;
            y = 0;
            vx = 1;
            vy = 1;
            while (true) {
                x += vx;
                y += vy;
                if ((x < 0) || (x > w - (h / 4))) {
                    vx = -vx;
                }
                if ((y < 0) || (y > h - (h / 4))) {
                    vy = -vy;
                }
                l1.setLayerPosition(x, y, 0);
```

```
                YAPI.Sleep(5);
            }

        } catch (YAPI_Exception ex) {
            System.out.println("Exception durring execution (" + ex.getLocalizedMessage() +
")");
            YAPI.FreeAPI();
            System.exit(1);
        }

    }
}
```

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.

```
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();
    }
}
```

17.4. 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.

18. Using the Yocto-MiniDisplay 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.

18.1. 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.

18.2. 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.

18.3. 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].

18.4. 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>
```

18.5. Control of the Display function

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

```
[...]

// Retrieving the object representing the module (connected here locally by USB)
YAPI.EnableUSBHost(this);
YAPI.RegisterHub("usb");
display = YDisplay.FindDisplay("YD096X16-123456.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-MiniDisplay module with serial number *YD096X16-123456* 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("YD096X16-123456.display")
display = YDisplay.FindDisplay("YD096X16-123456.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.

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

public class GettingStarted_Yocto_MiniDisplay extends Activity implements
OnItemSelectedListener
{


    private YDisplay display = null;
    private ArrayAdapter<String> aa;

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

    @Override
    protected void onStart()
    {
        super.onStart();
        aa.clear();
        try {
            YAPI.EnableUSBHost(this);
            YAPI.RegisterHub("usb");
            YDisplay d = YDisplay.FirstDisplay();
            while (d != null) {
                String hwid = d.get_hardwareId();
                aa.add(hwid);
                d = d.nextDisplay();
            }
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
        aa.notifyDataSetChanged();
    }

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

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

```
        updateDisplay(null);
    }

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

    public void updateDisplay(View view)
    {
        if (display == null)
            return;

        EditText message = (EditText) findViewById(R.id.editText1);
        // clean up
        try {
            display.resetAll();

            // retreive the display size
            int w = display.get_displayWidth();
            int h = display.get_displayHeight();

            // reteive the first layer
            YDisplayLayer l0 = display.get_displayLayer(0);

            // display a text in the middle of the screen
            l0.drawText(w / 2, h / 2, YDisplayLayer.ALIGN.CENTER, message.getText(
).toString());

            // visualize each corner
            l0.moveTo(0, 5);
            l0.lineTo(0, 0);
            l0.lineTo(5, 0);
            l0.moveTo(0, h - 6);
            l0.lineTo(0, h - 1);
            l0.lineTo(5, h - 1);
            l0.moveTo(w - 1, h - 6);
            l0.lineTo(w - 1, h - 1);
            l0.lineTo(w - 6, h - 1);
            l0.moveTo(w - 1, 5);
            l0.lineTo(w - 1, 0);
            l0.lineTo(w - 6, 0);
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }

    }


}
```

18.6. Control of the module part

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

```
package com.yoctopuce.doc_examples;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemSelectedListener;
import android.widget.ArrayAdapter;
import android.widget.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);
            Log.d("switch", "beacon" + module.get_beacon());
            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.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);
                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();
    }

}
```

18.7. Error handling

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

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

In the 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.

19. Advanced programming

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

19.1. 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.

20. Using with unsupported languages

Yoctopuce modules can be driven from most common programming languages. New languages are regularly added, depending on the interest expressed by Yoctopuce product users. Nevertheless, some languages are not, and will never be, supported by Yoctopuce. There can be several reasons for this: compilers which are not available anymore, unadapted environments, etc.

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

20.1. Command line

The easiest method to drive Yoctopuce modules from an unsupported programming language is to use the command line API through system calls. The command line API is in fact made of a group of small executables which are easy to call. Their output is also easy to analyze. As most programming languages allow you to make system calls, the issue is solved with a few lines of code.

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

20.2. 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-MiniDisplay with the *YD096X16-12345* 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/YD096X16-12345/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/YD096X16-12345/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/YD096X16-12345/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/YD096X16-12345/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/YD096X16-12345/api/display/logicalName
```

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

```
http://127.0.0.1:4444/bySerial/YD096X16-12345/api/display?logicalName=myFunction
```

You can find the list of available attributes for your Yocto-MiniDisplay 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/YD096X16-12345/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/YD096X16-12345/dataLogger.json?id=display&utc=1389801080
```

20.3. 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
libyapi-armhf.so	Linux ARM HL
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      = 'YD096X16-12345';
  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.
```

20.4. Porting the high level library

As all the sources of the Yoctopuce API are fully provided, you can very well port the whole API in the language of your choice. Note, however, that a large portion of the API source code is automatically generated.

Therefore, it is not necessary for you to port the complete API. You only need to port the *yocto_api* file and one file corresponding to a function, for example *yocto_relay*. After a little additional work, Yoctopuce is then able to generate all other files. Therefore, we highly recommend that you contact Yoctopuce support before undertaking to port the Yoctopuce library in another language. Collaborative work is advantageous to both parties.

21. High-level API Reference

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

21.1. 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:

```
js      <script type='text/javascript' src='yocto_api.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YAPI = yoctolib.YAPI;
        var YModule = yoctolib.YModule;
php     require_once('yocto_api.php');
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;
py      from yocto_api import *
```

Global functions
yCheckLogicalName(**name**)
Checks if a given string is valid as logical name for a module or a function.
yDisableExceptions()
Disables the use of exceptions to report runtime errors.
yEnableExceptions()
Re-enables the use of exceptions for runtime error handling.
yEnableUSBHost(**osContext**)
This function is used only on Android.
yFreeAPI()
Frees dynamically allocated memory blocks used by the Yoctopuce library.
yGetAPIVersion()
Returns the version identifier for the Yoctopuce library in use.
yGetTickCount()
Returns the current value of a monotone millisecond-based time counter.
yHandleEvents(**errmsg**)
Maintains the device-to-library communication channel.
yInitAPI(**mode**, **errmsg**)
Initializes the Yoctopuce programming library explicitly.
yPreregisterHub(**url**, **errmsg**)
Fault-tolerant alternative to RegisterHub().
yRegisterDeviceArrivalCallback(**arrivalCallback**)
Register a callback function, to be called each time a device is pluged.
yRegisterDeviceRemovalCallback(**removalCallback**)
Register a callback function, to be called each time a device is unpluged.
yRegisterHub(**url**, **errmsg**)
Setup the Yoctopuce library to use modules connected on a given machine.
yRegisterHubDiscoveryCallback(**callback**)

Register a callback function, to be called each time a network hub or a VirtualHub is detected on the local network.

yRegisterLogFunction(**logfun**)

Registers a log callback function.

ySelectArchitecture(**arch**)

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

ySetDelegate(**object**)

(Objective-C only) Register an object that must follow the procol `YDeviceHotPlug`.

ySetTimeout(**callback**, **ms_timeout**, **arguments**)

Invoke the specified callback function after a given timeout.

ySleep(**ms_duration**, **errmsg**)

Pauses the execution flow for a specified duration.

yUnregisterHub(**url**)

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

yUpdateDeviceList(**errmsg**)

Triggers a (re)detection of connected Yoctopuce modules.

yUpdateDeviceList_async(**callback**, **context**)

Triggers a (re)detection of connected Yoctopuce modules.

YAPI.CheckLogicalName() YAPI
yCheckLogicalName()

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

```
js      function yCheckLogicalName( name)
nodejs  function CheckLogicalName( name)
php     function yCheckLogicalName( $name)
cpp     bool yCheckLogicalName( const string& name)
m       BOOL yCheckLogicalName( NSString * name)
pas     function yCheckLogicalName( name: string): boolean
vb      function yCheckLogicalName( ByVal name As String) As Boolean
cs      bool CheckLogicalName( string name)
java    boolean CheckLogicalName( String name)
py      def 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.DisableExceptions() YAPI
yDisableExceptions()

Disables the use of exceptions to report runtime errors.

```
js      function yDisableExceptions( )
nodejs  function DisableExceptions( )
php     function yDisableExceptions( )
cpp     void yDisableExceptions( )
m       void yDisableExceptions( )
pas     procedure yDisableExceptions( )
vb      procedure yDisableExceptions( )
cs      void DisableExceptions( )
py      def 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
yEnableExceptions()

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

js	function **yEnableExceptions**( )
nodejs	function **EnableExceptions**( )
php	function **yEnableExceptions**( )
cpp	void **yEnableExceptions**( )
m	void **yEnableExceptions**( )
pas	procedure **yEnableExceptions**( )
vb	procedure **yEnableExceptions**( )
cs	void **EnableExceptions**( )
py	def **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
yEnableUSBHost()

This function is used only on Android.

java synchronized static 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 subclasee). 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
yFreeAPI()

Frees dynamically allocated memory blocks used by the Yoctopuce library.

js	function **yFreeAPI**( )
nodejs	function **FreeAPI**( )
php	function **yFreeAPI**( )
cpp	void **yFreeAPI**( )
m	void **yFreeAPI**( )
pas	procedure **yFreeAPI**( )
vb	procedure **yFreeAPI**( )
cs	void **FreeAPI**( )
java	synchronized static void **FreeAPI**( )
py	def **FreeAPI**( )

It is generally not required to call this function, unless you want to free all dynamically allocated memory blocks in order to track a memory leak for instance. You should not call any other library function after calling `yFreeAPI()`, or your program will crash.

YAPI.GetAPIVersion() YAPI
yGetAPIVersion()

Returns the version identifier for the Yoctopuce library in use.

```
js      function yGetAPIVersion( )
nodejs  function GetAPIVersion( )
php     function yGetAPIVersion( )
cpp     string yGetAPIVersion( )
m       NSString* yGetAPIVersion( )
pas     function yGetAPIVersion( ): string
vb      function yGetAPIVersion( ) As String
cs      String GetAPIVersion( )
java    String GetAPIVersion( )
py      def 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.GetTickCount() YAPI
yGetTickCount()

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

```
js      function yGetTickCount( )
nodejs  function GetTickCount( )
php     function yGetTickCount( )
cpp     u64 yGetTickCount( )
m       u64 yGetTickCount( )
pas     function yGetTickCount( ): u64
vb      function yGetTickCount( ) As Long
cs      ulong GetTickCount( )
java    long GetTickCount( )
py      def 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
yHandleEvents()

Maintains the device-to-library communication channel.

```
js      function yHandleEvents( errmsg)
nodejs  function HandleEvents( errmsg)
php     function yHandleEvents( &$errmsg)
cpp     YRETCODE yHandleEvents( string& errmsg)
m       YRETCODE yHandleEvents( NSError** errmsg)
pas     function yHandleEvents( var errmsg: string): integer
vb      function yHandleEvents( ByRef errmsg As String) As YRETCODE
cs      YRETCODE HandleEvents( ref string errmsg)
java    int HandleEvents( )
py      def HandleEvents( errmsg=None)
```

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
yInitAPI()

Initializes the Yoctopuce programming library explicitly.

```
js      function yInitAPI( mode, errmsg)
nodejs  function InitAPI( mode, errmsg)
php     function yInitAPI( $mode, &$errmsg)
cpp     YRETCODE yInitAPI( int mode, string& errmsg)
m       YRETCODE yInitAPI( int mode, NSError** errmsg)
pas     function yInitAPI( mode: integer, var errmsg: string): integer
vb      function yInitAPI( ByVal mode As Integer, ByRef errmsg As String) As Integer
cs      int InitAPI( int mode, ref string errmsg)
java    synchronized static int InitAPI( int mode)
py      def InitAPI( mode, errmsg=None)
```

It is not strictly needed to call `yInitAPI()`, as the library is automatically initialized when calling `yRegisterHub()` for the first time.

When `Y_DETECT_NONE` is used as detection `mode`, you must explicitly use `yRegisterHub()` to point the API to the VirtualHub on which your devices are connected before trying to access them.

Parameters :

mode an integer corresponding to the type of automatic device detection to use. Possible values are `Y_DETECT_NONE`, `Y_DETECT_USB`, `Y_DETECT_NET`, and `Y_DETECT_ALL`.

errmsg a string passed by reference to receive any error message.

Returns :

`YAPI_SUCCESS` when the call succeeds. On failure, throws an exception or returns a negative error code.

YAPI.PreregisterHub() YAPI
yPreregisterHub()

Fault-tolerant alternative to RegisterHub().

```
js      function yPreregisterHub( url, errmsg)
nodejs  function PreregisterHub( url, errmsg)
php     function yPreregisterHub( $url, &$errmsg)
cpp     YRETCODE yPreregisterHub( const string& url, string& errmsg)
m       YRETCODE yPreregisterHub( NSString * url, NSError** errmsg)
pas     function yPreregisterHub( url: string, var errmsg: string): integer
vb      function yPreregisterHub( ByVal url As String,
                                  ByRef errmsg As String) As Integer
cs      int PreregisterHub( string url, ref string errmsg)
java    synchronized static int PreregisterHub( String url)
py      def PreregisterHub( url, errmsg=None)
```

This function has the same purpose and same arguments as `RegisterHub()`, 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 yRegisterDeviceArrivalCallback()

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

js	function **yRegisterDeviceArrivalCallback**(**arrivalCallback**)
nodejs	function **RegisterDeviceArrivalCallback**(**arrivalCallback**)
php	function **yRegisterDeviceArrivalCallback**(**$arrivalCallback**)
cpp	void **yRegisterDeviceArrivalCallback**(yDeviceUpdateCallback **arrivalCallback**)
m	void **yRegisterDeviceArrivalCallback**(yDeviceUpdateCallback **arrivalCallback**)
pas	procedure **yRegisterDeviceArrivalCallback**(**arrivalCallback**: yDeviceUpdateFunc)
vb	procedure **yRegisterDeviceArrivalCallback**(ByVal **arrivalCallback** As yDeviceUpdateFunc)
cs	void **RegisterDeviceArrivalCallback**(yDeviceUpdateFunc **arrivalCallback**)
java	synchronized static void **RegisterDeviceArrivalCallback**(DeviceArrivalCallback **arrivalCallback**)
py	def **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() yRegisterDeviceRemovalCallback()

YAPI

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

js	function **yRegisterDeviceRemovalCallback**(**removalCallback**)
nodejs	function **RegisterDeviceRemovalCallback**(**removalCallback**)
php	function **yRegisterDeviceRemovalCallback**(**$removalCallback**)
cpp	void **yRegisterDeviceRemovalCallback**(yDeviceUpdateCallback **removalCallback**)
m	void **yRegisterDeviceRemovalCallback**(yDeviceUpdateCallback **removalCallback**)
pas	procedure **yRegisterDeviceRemovalCallback**(**removalCallback**: yDeviceUpdateFunc)
vb	procedure **yRegisterDeviceRemovalCallback**(ByVal **removalCallback** As yDeviceUpdateFunc)
cs	void **RegisterDeviceRemovalCallback**(yDeviceUpdateFunc **removalCallback**)
java	synchronized static void **RegisterDeviceRemovalCallback**(DeviceRemovalCallback **removalCallback**)
py	def **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
yRegisterHub()

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

```
js      function yRegisterHub( url, errmsg)
nodejs  function RegisterHub( url, errmsg)
php     function yRegisterHub( $url, &$errmsg)
cpp     YRETCODE yRegisterHub( const string& url, string& errmsg)
m       YRETCODE yRegisterHub( NSString * url, NSError** errmsg)
pas     function yRegisterHub( url: string, var errmsg: string): integer
vb      function yRegisterHub( ByVal url As String,
                               ByRef errmsg As String) As Integer
cs      int RegisterHub( string url, ref string errmsg)
java    synchronized static int RegisterHub( String url)
py      def RegisterHub( url, errmsg=None)
```

The parameter will determine how the API will work. Use the follwing 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 harware, 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 keywork 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 ou 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 acces control has been activated on the hub, virtual or not, you want to reach, the URL parameter should look like:

```
http://username:password@adresse: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 yRegisterHubDiscoveryCallback()

Register a callback function, to be called each time a network hub or a VirtualHub is detected on the local network.

java void **RegisterHubDiscoveryCallback**(NewHubCallback **callback**)

Parameters :

callback a procedure taking a two string as parameter, or null

YAPI.RegisterLogFunction() YAPI
yRegisterLogFunction()

Registers a log callback function.

cpp	void **yRegisterLogFunction**(yLogFunction **logfun**)
m	void **yRegisterLogFunction**(yLogCallback **logfun**)
pas	procedure **yRegisterLogFunction**(**logfun**: yLogFunc)
vb	procedure **yRegisterLogFunction**(ByVal **logfun** As yLogFunc)
cs	void **RegisterLogFunction**(yLogFunc **logfun**)
java	void **RegisterLogFunction**(LogCallback **logfun**)
py	def **RegisterLogFunction**(**logfun**)

This callback will be called each time the API have something to say. Quite usefull to debug the API.

Parameters :

logfun a procedure taking a string parameter, or `null`

YAPI.SelectArchitecture() YAPI ySelectArchitecture()

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

py def **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"`, `"i386"`,`"x86_64"`,`"32bit"`, `"64bit"`

Returns :

nothing.

On failure, throws an exception.

YAPI.SetDelegate() YAPI ySetDelegate()

(Objective-C only) Register an object that must follow the procol YDeviceHotPlug.

m void **ySetDelegate**(id **object**)

The methodes 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 procol YAPIDelegate, or nil

YAPI.SetTimeout() YAPI
ySetTimeout()

Invoke the specified callback function after a given timeout.

```
js      function ySetTimeout( callback, ms_timeout, arguments)
nodejs  function SetTimeout( callback, ms_timeout, arguments)
```

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.
arguments	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.Sleep() YAPI
ySleep()

Pauses the execution flow for a specified duration.

```
js      function ySleep( ms_duration, errmsg)
nodejs  function Sleep( ms_duration, errmsg)
php     function ySleep( $ms_duration, &$errmsg)
cpp     YRETCODE ySleep( unsigned ms_duration, string& errmsg)
m       YRETCODE ySleep( unsigned ms_duration, NSError ** errmsg)
pas     function ySleep( ms_duration: integer, var errmsg: string): integer
vb      function ySleep( ByVal ms_duration As Integer,
                         ByRef errmsg As String) As Integer
cs      int Sleep( int ms_duration, ref string errmsg)
java    int Sleep( long ms_duration)
py      def Sleep( ms_duration, errmsg=None)
```

This function implements a passive waiting loop, meaning that it does not consume CPU cycles significatively. 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.UnregisterHub() YAPI yUnregisterHub()

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

```
js      function yUnregisterHub( url)
nodejs  function UnregisterHub( url)
php     function yUnregisterHub( $url)
cpp     void yUnregisterHub( const string& url)
m       void yUnregisterHub( NSString * url)
pas     procedure yUnregisterHub( url: string)
vb      procedure yUnregisterHub( ByVal url As String)
cs      void UnregisterHub( string url)
java    synchronized static void UnregisterHub( String url)
py      def UnregisterHub( url)
```

Parameters :

url a string containing either **"usb"** or the

YAPI.UpdateDeviceList() YAPI yUpdateDeviceList()

Triggers a (re)detection of connected Yoctopuce modules.

```
js      function yUpdateDeviceList( errmsg)
nodejs  function UpdateDeviceList( errmsg)
php     function yUpdateDeviceList( &$errmsg)
cpp     YRETCODE yUpdateDeviceList( string& errmsg)
m       YRETCODE yUpdateDeviceList( NSError** errmsg)
pas     function yUpdateDeviceList( var errmsg: string): integer
vb      function yUpdateDeviceList( ByRef errmsg As String) As YRETCODE
cs      YRETCODE UpdateDeviceList( ref string errmsg)
java    int UpdateDeviceList( )
py      def UpdateDeviceList( errmsg=None)
```

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.

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
yUpdateDeviceList_async()

Triggers a (re)detection of connected Yoctopuce modules.

```
js      function yUpdateDeviceList_async( callback, context)
nodejs  function UpdateDeviceList_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.

21.2. Module control interface

This interface is identical for all Yoctopuce USB modules. 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>
nodejs  var yoctolib = require('yoctolib');
        var YAPI = yoctolib.YAPI;
        var YModule = yoctolib.YModule;
php     require_once('yocto_api.php');
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;
py      from yocto_api import *
```

Global functions
yFindModule(**func**) Allows you to find a module from its serial number or from its logical name.
yFirstModule() Starts the enumeration of modules currently accessible.
`YModule` methods
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→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→functionValue(**functionIndex**) Retrieves the advertised value of the *n*th function on 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_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_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 hardware release version of the module.

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_upTime()

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

module→get_usbBandwidth()

Returns the number of USB interfaces used by 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→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→nextModule()

Continues the module enumeration started using `yFirstModule()`.

module→reboot(**secBeforeReboot**)

Schedules a simple module reboot after the given number of seconds.

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_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_usbBandwidth(newval)

Changes the number of USB interfaces used by the module.

module→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

module→triggerFirmwareUpdate(secBeforeReboot)

Schedules a module reboot into special firmware update mode.

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
yFindModule()

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

```
js      function yFindModule( func)
nodejs  function FindModule( func)
php     function yFindModule( $func)
cpp     YModule* yFindModule( string func)
m       +(YModule*) yFindModule : (NSString*) func
pas     function yFindModule( func: string): TYModule
vb      function yFindModule( ByVal func As String) As YModule
cs      YModule FindModule( string func)
java    YModule FindModule( String func)
py      def FindModule( 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.

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.FirstModule() YModule
yFirstModule()

Starts the enumeration of modules currently accessible.

```
js      function yFirstModule( )
nodejs  function FirstModule( )
php     function yFirstModule( )
cpp     YModule* yFirstModule( )
m       YModule* yFirstModule( )
pas     function yFirstModule( ): TYModule
vb      function yFirstModule( ) As YModule
cs      YModule FirstModule( )
java    YModule FirstModule( )
py      def 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→describe() YModule

Returns a descriptive text that identifies the module.

```
js      function describe( )
nodejs  function describe( )
php     function describe( )
cpp     string describe( )
m       -(NSString*) describe
pas     function describe( ): string
vb      function describe( ) As String
cs      string describe( )
java    String describe( )
py      def 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)
nodejs  function download( pathname)
php     function download( $pathname)
cpp     string download( string pathname)
m       -(NSData*) download : (NSString*) pathname
pas     function download( pathname: string): TByteArray
vb      function download( ) As Byte
py      def download( 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 an empty content.

module→functionCount() YModule

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

```
js      function functionCount( )
nodejs  function functionCount( )
php     function functionCount( )
cpp     int functionCount( )
m       -(int) functionCount
pas     function functionCount( ): integer
vb      function functionCount( ) As Integer
cs      int functionCount( )
py      def 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)
nodejs  function functionId( functionIndex)
php     function functionId( $functionIndex)
cpp     string functionId( int functionIndex)
m       -(NSString*) functionId : (int) functionIndex
pas     function functionId( functionIndex: integer): string
vb      function functionId( ByVal functionIndex As Integer) As String
cs      string functionId( int functionIndex)
py      def 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)
nodejs  function functionName( functionIndex)
php     function functionName( $functionIndex)
cpp     string functionName( int functionIndex)
m       -(NSString*) functionName : (int) functionIndex
pas     function functionName( functionIndex: integer): string
vb      function functionName( ByVal functionIndex As Integer) As String
cs      string functionName( int functionIndex)
py      def 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→functionValue() YModule

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

```
js      function functionValue( functionIndex)
nodejs  function functionValue( functionIndex)
php     function functionValue( $functionIndex)
cpp     string functionValue( int functionIndex)
m       -(NSString*) functionValue : (int) functionIndex
pas     function functionValue( functionIndex: integer): string
vb      function functionValue( ByVal functionIndex As Integer) As String
cs      string functionValue( int functionIndex)
py      def 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_beacon() YModule
module→beacon()

Returns the state of the localization beacon.

```
js      function get_beacon( )
nodejs  function get_beacon( )
php     function get_beacon( )
cpp     Y_BEACON_enum get_beacon( )
m       -(Y_BEACON_enum) beacon
pas     function get_beacon( ): Integer
vb      function get_beacon( ) As Integer
cs      int get_beacon( )
java    int get_beacon( )
py      def get_beacon( )
cmd     YModule target get_beacon
```

Returns :

either Y_BEACON_OFF or Y_BEACON_ON, according to the state of the localization beacon

On failure, throws an exception or returns Y_BEACON_INVALID.

module→get_errorMessage() YModule
module→errorMessage()

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

```
js      function get_errorMessage( )
nodejs  function get_errorMessage( )
php     function get_errorMessage( )
cpp     string get_errorMessage( )
m       -(NSString*) errorMessage
pas     function get_errorMessage( ): string
vb      function get_errorMessage( ) As String
cs      string get_errorMessage( )
java    String get_errorMessage( )
py      def 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 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( )
nodejs  function get_errorType( )
php     function get_errorType( )
cpp     YRETCODE get_errorType( )
pas     function get_errorType( ): YRETCODE
vb      function get_errorType( ) As YRETCODE
cs      YRETCODE get_errorType( )
java    int get_errorType( )
py      def 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 occured 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( )
nodejs  function get_firmwareRelease( )
php     function get_firmwareRelease( )
cpp     string get_firmwareRelease( )
m       -(NSString*) firmwareRelease
pas     function get_firmwareRelease( ): string
vb      function get_firmwareRelease( ) As String
cs      string get_firmwareRelease( )
java    String get_firmwareRelease( )
py      def get_firmwareRelease( )
cmd     YModule target get_firmwareRelease
```

Returns :

a string corresponding to the version of the firmware embedded in the module

On failure, throws an exception or returns `Y_FIRMWARERELEASE_INVALID`.

module→get_hardwareId() YModule
module→hardwareId()

Returns the unique hardware identifier of the module.

```
js      function get_hardwareId( )
nodejs  function get_hardwareId( )
php     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      def 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( )
nodejs  function get_icon2d( )
php     function get_icon2d( )
cpp     string get_icon2d( )
m       -(NSData*) icon2d
pas     function get_icon2d( ): TByteArray
vb      function get_icon2d( ) As Byte
py      def 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.

module→get_lastLogs() YModule
module→lastLogs()

Returns a string with last logs of the module.

```
js      function get_lastLogs( )
nodejs  function get_lastLogs( )
php     function get_lastLogs( )
cpp     string get_lastLogs( )
m       -(NSString*) lastLogs
pas     function get_lastLogs( ): string
vb      function get_lastLogs( ) As String
cs      string get_lastLogs( )
java    String get_lastLogs( )
py      def 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.

module→get_logicalName() YModule
module→logicalName()

Returns the logical name of the module.

```
js      function get_logicalName( )
nodejs  function get_logicalName( )
php     function get_logicalName( )
cpp     string get_logicalName( )
m       -(NSString*) logicalName
pas     function get_logicalName( ): string
vb      function get_logicalName( ) As String
cs      string get_logicalName( )
java    String get_logicalName( )
py      def get_logicalName( )
cmd     YModule target get_logicalName
```

Returns :
a string corresponding to the logical name of the module

On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

module→get_luminosity() YModule
module→luminosity()

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

```
js      function get_luminosity( )
nodejs  function get_luminosity( )
php     function get_luminosity( )
cpp     int get_luminosity( )
m       -(int) luminosity
pas     function get_luminosity( ): LongInt
vb      function get_luminosity( ) As Integer
cs      int get_luminosity( )
java    int get_luminosity( )
py      def get_luminosity( )
cmd     YModule target get_luminosity
```

Returns :
an integer corresponding to the luminosity of the module informative leds (from 0 to 100)

On failure, throws an exception or returns `Y_LUMINOSITY_INVALID`.

module→get_persistentSettings() YModule
module→persistentSettings()

Returns the current state of persistent module settings.

```
js      function get_persistentSettings( )
nodejs  function get_persistentSettings( )
php     function get_persistentSettings( )
cpp     Y_PERSISTENTSETTINGS_enum get_persistentSettings( )
m       -(Y_PERSISTENTSETTINGS_enum) persistentSettings
pas     function get_persistentSettings( ): Integer
vb      function get_persistentSettings( ) As Integer
cs      int get_persistentSettings( )
java    int get_persistentSettings( )
py      def get_persistentSettings( )
cmd     YModule target get_persistentSettings
```

Returns :

a value among `Y_PERSISTENTSETTINGS_LOADED`, `Y_PERSISTENTSETTINGS_SAVED` and `Y_PERSISTENTSETTINGS_MODIFIED` corresponding to the current state of persistent module settings

On failure, throws an exception or returns `Y_PERSISTENTSETTINGS_INVALID`.

module→get_productId() YModule
module→productId()

Returns the USB device identifier of the module.

```
js      function get_productId( )
nodejs  function get_productId( )
php     function get_productId( )
cpp     int get_productId( )
m       -(int) productId
pas     function get_productId( ): LongInt
vb      function get_productId( ) As Integer
cs      int get_productId( )
java    int get_productId( )
py      def get_productId( )
cmd     YModule target get_productId
```

Returns :
an integer corresponding to the USB device identifier of the module

On failure, throws an exception or returns `Y_PRODUCTID_INVALID`.

module→get_productName() YModule
module→productName()

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

```
js      function get_productName( )
nodejs  function get_productName( )
php     function get_productName( )
cpp     string get_productName( )
m       -(NSString*) productName
pas     function get_productName( ): string
vb      function get_productName( ) As String
cs      string get_productName( )
java    String get_productName( )
py      def get_productName( )
cmd     YModule target get_productName
```

Returns :
a string corresponding to the commercial name of the module, as set by the factory

On failure, throws an exception or returns `Y_PRODUCTNAME_INVALID`.

module→get_productRelease() YModule
module→productRelease()

Returns the hardware release version of the module.

```
js      function get_productRelease( )
nodejs  function get_productRelease( )
php     function get_productRelease( )
cpp     int get_productRelease( )
m       -(int) productRelease
pas     function get_productRelease( ): LongInt
vb      function get_productRelease( ) As Integer
cs      int get_productRelease( )
java    int get_productRelease( )
py      def get_productRelease( )
cmd     YModule target get_productRelease
```

Returns :
an integer corresponding to the hardware release version of the module

On failure, throws an exception or returns `Y_PRODUCTRELEASE_INVALID`.

module→get_rebootCountdown() YModule
module→rebootCountdown()

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

js	function **get_rebootCountdown**( )
nodejs	function **get_rebootCountdown**( )
php	function **get_rebootCountdown**( )
cpp	int **get_rebootCountdown**( )
m	-(int) rebootCountdown
pas	function **get_rebootCountdown**( ): LongInt
vb	function **get_rebootCountdown**( ) As Integer
cs	int **get_rebootCountdown**( )
java	int **get_rebootCountdown**( )
py	def **get_rebootCountdown**( )
cmd	YModule **target get_rebootCountdown**

Returns :

an integer corresponding to the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled

On failure, throws an exception or returns `Y_REBOOTCOUNTDOWN_INVALID`.

module→get_serialNumber() YModule
module→serialNumber()

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

```
js      function get_serialNumber( )
nodejs  function get_serialNumber( )
php     function get_serialNumber( )
cpp     string get_serialNumber( )
m       -(NSString*) serialNumber
pas     function get_serialNumber( ): string
vb      function get_serialNumber( ) As String
cs      string get_serialNumber( )
java    String get_serialNumber( )
py      def get_serialNumber( )
cmd     YModule target get_serialNumber
```

Returns :

a string corresponding to the serial number of the module, as set by the factory

On failure, throws an exception or returns `Y_SERIALNUMBER_INVALID`.

module→get_upTime() YModule
module→upTime()

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

```
js      function get_upTime( )
nodejs  function get_upTime( )
php     function get_upTime( )
cpp     s64 get_upTime( )
m       -(s64) upTime
pas     function get_upTime( ): int64
vb      function get_upTime( ) As Long
cs      long get_upTime( )
java    long get_upTime( )
py      def get_upTime( )
cmd     YModule target get_upTime
```

Returns :

an integer corresponding to the number of milliseconds spent since the module was powered on

On failure, throws an exception or returns `Y_UPTIME_INVALID`.

module→get_usbBandwidth() YModule
module→usbBandwidth()

Returns the number of USB interfaces used by the module.

js	function **get_usbBandwidth**( )
nodejs	function **get_usbBandwidth**( )
php	function **get_usbBandwidth**( )
cpp	Y_USBBANDWIDTH_enum **get_usbBandwidth**( )
m	-(Y_USBBANDWIDTH_enum) usbBandwidth
pas	function **get_usbBandwidth**( ): Integer
vb	function **get_usbBandwidth**( ) As Integer
cs	int **get_usbBandwidth**( )
java	int **get_usbBandwidth**( )
py	def **get_usbBandwidth**( )
cmd	YModule **target get_usbBandwidth**

Returns :

either `Y_USBBANDWIDTH_SIMPLE` or `Y_USBBANDWIDTH_DOUBLE`, according to the number of USB interfaces used by the module

On failure, throws an exception or returns `Y_USBBANDWIDTH_INVALID`.

module→get_usbCurrent() YModule
module→usbCurrent()

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

```
js      function get_usbCurrent( )
nodejs  function get_usbCurrent( )
php     function get_usbCurrent( )
cpp     int get_usbCurrent( )
m       -(int) usbCurrent
pas     function get_usbCurrent( ): LongInt
vb      function get_usbCurrent( ) As Integer
cs      int get_usbCurrent( )
java    int get_usbCurrent( )
py      def get_usbCurrent( )
cmd     YModule target get_usbCurrent
```

Returns :

an integer corresponding to the current consumed by the module on the USB bus, in milli-amps

On failure, throws an exception or returns `Y_USBCURRENT_INVALID`.

module→get_userData() YModule
module→userData()

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

```
js      function get_userData( )
nodejs  function get_userData( )
php     function get_userData( )
cpp     void * get_userData( )
m       -(void*) userData
pas     function get_userData( ): Tobject
vb      function get_userData( ) As Object
cs      object get_userData( )
java    Object get_userData( )
py      def 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→isOnline() YModule

Checks if the module is currently reachable, without raising any error.

```
js      function isOnline( )
nodejs  function isOnline( )
php     function isOnline( )
cpp     bool isOnline( )
m       -(BOOL) isOnline
pas     function isOnline( ): boolean
vb      function isOnline( ) As Boolean
cs      bool isOnline( )
java    boolean isOnline( )
py      def 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)
nodejs  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)
nodejs  function load( msValidity)
php     function load( $msValidity)
cpp     YRETCODE load( int msValidity)
m       -(YRETCODE) load : (int) msValidity
pas     function load( msValidity: integer): YRETCODE
vb      function load( ByVal msValidity As Integer) As YRETCODE
cs      YRETCODE load( int msValidity)
java    int load( long msValidity)
py      def 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 trafic 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)
nodejs  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 trafic 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→nextModule() YModule

Continues the module enumeration started using `yFirstModule()`.

```
js      function nextModule( )
nodejs  function nextModule( )
php     function nextModule( )
cpp     YModule * nextModule( )
m       -(YModule*) nextModule
pas     function nextModule( ): TYModule
vb      function nextModule( ) As YModule
cs      YModule nextModule( )
java    YModule nextModule( )
py      def nextModule( )
```

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)
nodejs  function reboot( secBeforeReboot)
php     function reboot( $secBeforeReboot)
cpp     int reboot( int secBeforeReboot)
m       -(int) reboot : (int) secBeforeReboot
pas     function reboot( secBeforeReboot: LongInt): LongInt
vb      function reboot( ) As Integer
cs      int reboot( int secBeforeReboot)
java    int reboot( int secBeforeReboot)
py      def reboot( 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→revertFromFlash() YModule

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

```
js      function revertFromFlash( )
nodejs  function revertFromFlash( )
php     function revertFromFlash( )
cpp     int revertFromFlash( )
m       -(int) revertFromFlash
pas     function revertFromFlash( ): LongInt
vb      function revertFromFlash( ) As Integer
cs      int revertFromFlash( )
java    int revertFromFlash( )
py      def 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( )
nodejs  function saveToFlash( )
php     function saveToFlash( )
cpp     int saveToFlash( )
m       -(int) saveToFlash
pas     function saveToFlash( ): LongInt
vb      function saveToFlash( ) As Integer
cs      int saveToFlash( )
java    int saveToFlash( )
py      def 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_beacon() YModule
module→setBeacon()

Turns on or off the module localization beacon.

js	function **set_beacon**(**newval**)
nodejs	function **set_beacon**(**newval**)
php	function **set_beacon**(**$newval**)
cpp	int **set_beacon**(Y_BEACON_enum **newval**)
m	-(int) setBeacon : (Y_BEACON_enum) **newval**
pas	function **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**)
py	def **set_beacon**(**newval**)
cmd	YModule **target set_beacon newval**

Parameters :

newval either `Y_BEACON_OFF` or `Y_BEACON_ON`

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_logicalName() YModule
module→setLogicalName()

Changes the logical name of the module.

```
js      function set_logicalName( newval)
nodejs  function set_logicalName( newval)
php     function set_logicalName( $newval)
cpp     int set_logicalName( const string& newval)
m       -(int) setLogicalName : (NSString*) newval
pas     function 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)
py      def set_logicalName( 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)
nodejs  function set_luminosity( newval)
php     function set_luminosity( $newval)
cpp     int set_luminosity( int newval)
m       -(int) setLuminosity : (int) newval
pas     function 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)
py      def set_luminosity( 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_usbBandwidth() YModule
module→setUsbBandwidth()

Changes the number of USB interfaces used by the module.

```
js      function set_usbBandwidth( newval)
nodejs  function set_usbBandwidth( newval)
php     function set_usbBandwidth( $newval)
cpp     int set_usbBandwidth( Y_USBBANDWIDTH_enum newval)
m       -(int) setUsbBandwidth : (Y_USBBANDWIDTH_enum) newval
pas     function set_usbBandwidth( newval: Integer): integer
vb      function set_usbBandwidth( ByVal newval As Integer) As Integer
cs      int set_usbBandwidth( int newval)
java    int set_usbBandwidth( int newval)
py      def set_usbBandwidth( newval)
cmd     YModule target set_usbBandwidth newval
```

You must reboot the module after changing this setting.

Parameters :

newval either `Y_USBBANDWIDTH_SIMPLE` or `Y_USBBANDWIDTH_DOUBLE`, according to the number of USB interfaces used by the module

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

module→set_userData() YModule
module→setUserData()

Stores a user context provided as argument in the userData attribute of the function.

```
js      function set_userData( data)
nodejs  function set_userData( data)
php     function set_userData( $data)
cpp     void set_userData( void* data)
m       -(void) setUserData : (void*) data
pas     procedure 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      def 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→triggerFirmwareUpdate() YModule

Schedules a module reboot into special firmware update mode.

```
js      function triggerFirmwareUpdate( secBeforeReboot)
nodejs  function triggerFirmwareUpdate( secBeforeReboot)
php     function triggerFirmwareUpdate( $secBeforeReboot)
cpp     int triggerFirmwareUpdate( int secBeforeReboot)
m       -(int) triggerFirmwareUpdate : (int) secBeforeReboot
pas     function triggerFirmwareUpdate( secBeforeReboot: LongInt): LongInt
vb      function triggerFirmwareUpdate( ) As Integer
cs      int triggerFirmwareUpdate( int secBeforeReboot)
java    int triggerFirmwareUpdate( int secBeforeReboot)
py      def triggerFirmwareUpdate( 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→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)
nodejs  function 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.

21.3. Display function interface

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 use the functions described here, you should include:

```
js      <script type='text/javascript' src='yocto_display.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YDisplay = yoctolib.YDisplay;
php     require_once('yocto_display.php');
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;
py      from yocto_display import *
```

Global functions
yFindDisplay(**func**) Retrieves a display for a given identifier.
yFirstDisplay() Starts the enumeration of displays currently accessible.
`YDisplay` methods
display→copyLayerContent(**srcLayerId**, **dstLayerId**) Copies the whole content of a layer to another layer.
display→describe() Returns a short text that describes 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_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→load(**msValidity**)

Preloads the display cache with a specified validity duration.

display→load_async(**msValidity**, **callback**, **context**)

Preloads the display cache with a specified validity duration (asynchronous version).

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→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 yFindDisplay()

Retrieves a display for a given identifier.

```
js      function yFindDisplay( func)
nodejs  function FindDisplay( func)
php     function yFindDisplay( $func)
cpp     YDisplay* yFindDisplay( string func)
m       +(YDisplay*) yFindDisplay : (NSString*) func
pas     function yFindDisplay( func: string): TYDisplay
vb      function yFindDisplay( ByVal func As String) As YDisplay
cs      YDisplay FindDisplay( string func)
java    YDisplay FindDisplay( String func)
py      def FindDisplay( 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 :

func a string that uniquely characterizes the display

Returns :

a `YDisplay` object allowing you to drive the display.

YDisplay.FirstDisplay() YDisplay yFirstDisplay()

Starts the enumeration of displays currently accessible.

```
js      function yFirstDisplay( )
nodejs  function FirstDisplay( )
php     function yFirstDisplay( )
cpp     YDisplay* yFirstDisplay( )
m       YDisplay* yFirstDisplay( )
pas     function yFirstDisplay( ): TYDisplay
vb      function yFirstDisplay( ) As YDisplay
cs      YDisplay FirstDisplay( )
java    YDisplay FirstDisplay( )
py      def 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.

display→copyLayerContent() YDisplay

Copies the whole content of a layer to another layer.

```
js      function copyLayerContent( srcLayerId, dstLayerId)
nodejs  function copyLayerContent( srcLayerId, dstLayerId)
php     function copyLayerContent( $srcLayerId, $dstLayerId)
cpp     int copyLayerContent( int srcLayerId, int dstLayerId)
m       -(int) copyLayerContent : (int) srcLayerId
                                : (int) dstLayerId
pas     function copyLayerContent( srcLayerId: LongInt,
                                   dstLayerId: LongInt): LongInt
vb      function copyLayerContent( ) As Integer
cs      int copyLayerContent( int srcLayerId, int dstLayerId)
java    int copyLayerContent( int srcLayerId, int dstLayerId)
py      def copyLayerContent( srcLayerId, 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 the display in the form TYPE(NAME)=SERIAL.FUNCTIONID.

```
js      function describe( )
nodejs  function describe( )
php     function describe( )
cpp     string describe( )
m       -(NSString*) describe
pas     function describe( ): string
vb      function describe( ) As String
cs      string describe( )
java    String describe( )
py      def 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)
nodejs  function fade( brightness, duration)
php     function fade( $brightness, $duration)
cpp     int fade( int brightness, int duration)
m       -(int) fade : (int) brightness : (int) duration
pas     function fade( brightness: LongInt, duration: LongInt): LongInt
vb      function fade( ) As Integer
cs      int fade( int brightness, int duration)
java    int fade( int brightness, int duration)
py      def fade( brightness, 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() YDisplay
display→advertisedValue()

Returns the current value of the display (no more than 6 characters).

```
js      function get_advertisedValue( )
nodejs  function get_advertisedValue( )
php     function get_advertisedValue( )
cpp     string get_advertisedValue( )
m       -(NSString*) advertisedValue
pas     function get_advertisedValue( ): string
vb      function get_advertisedValue( ) As String
cs      string get_advertisedValue( )
java    String get_advertisedValue( )
py      def 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 `Y_ADVERTISEDVALUE_INVALID`.

display→get_brightness() YDisplay
display→brightness()

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

```
js      function get_brightness( )
nodejs  function get_brightness( )
php     function get_brightness( )
cpp     int get_brightness( )
m       -(int) brightness
pas     function get_brightness( ): LongInt
vb      function get_brightness( ) As Integer
cs      int get_brightness( )
java    int get_brightness( )
py      def 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 `Y_BRIGHTNESS_INVALID`.

display→get_displayHeight() YDisplay
display→displayHeight()

Returns the display height, in pixels.

```
js      function get_displayHeight( )
nodejs  function get_displayHeight( )
php     function get_displayHeight( )
cpp     int get_displayHeight( )
m       -(int) displayHeight
pas     function get_displayHeight( ): LongInt
vb      function get_displayHeight( ) As Integer
cs      int get_displayHeight( )
java    int get_displayHeight( )
py      def get_displayHeight( )
cmd     YDisplay target get_displayHeight
```

Returns :
an integer corresponding to the display height, in pixels

On failure, throws an exception or returns `Y_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)
nodejs  function get_displayLayer( layerId)
php     function get_displayLayer( $layerId)
cpp     YDisplayLayer* get_displayLayer( unsigned layerId)
m       -(YDisplayLayer*) displayLayer : (unsigned) layerId
vb      function get_displayLayer( ) As YDisplayLayer
cs      YDisplayLayer get_displayLayer( int layerId)
java    synchronized YDisplayLayer 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() YDisplay
display→displayType()

Returns the display type: monochrome, gray levels or full color.

```
js      function get_displayType( )
nodejs  function get_displayType( )
php     function get_displayType( )
cpp     Y_DISPLAYTYPE_enum get_displayType( )
m       -(Y_DISPLAYTYPE_enum) displayType
pas     function get_displayType( ): Integer
vb      function get_displayType( ) As Integer
cs      int get_displayType( )
java    int get_displayType( )
py      def get_displayType( )
cmd     YDisplay target get_displayType
```

Returns :

a value among `Y_DISPLAYTYPE_MONO`, `Y_DISPLAYTYPE_GRAY` and `Y_DISPLAYTYPE_RGB` corresponding to the display type: monochrome, gray levels or full color

On failure, throws an exception or returns `Y_DISPLAYTYPE_INVALID`.

display→get_displayWidth() YDisplay
display→displayWidth()

Returns the display width, in pixels.

```
js      function get_displayWidth( )
nodejs  function get_displayWidth( )
php     function get_displayWidth( )
cpp     int get_displayWidth( )
m       -(int) displayWidth
pas     function get_displayWidth( ): LongInt
vb      function get_displayWidth( ) As Integer
cs      int get_displayWidth( )
java    int get_displayWidth( )
py      def get_displayWidth( )
cmd     YDisplay target get_displayWidth
```

Returns :
an integer corresponding to the display width, in pixels

On failure, throws an exception or returns `Y_DISPLAYWIDTH_INVALID`.

display→get_enabled() YDisplay
display→enabled()

Returns true if the screen is powered, false otherwise.

```
js      function get_enabled( )
nodejs  function get_enabled( )
php     function get_enabled( )
cpp     Y_ENABLED_enum get_enabled( )
m       -(Y_ENABLED_enum) enabled
pas     function get_enabled( ): Integer
vb      function get_enabled( ) As Integer
cs      int get_enabled( )
java    int get_enabled( )
py      def get_enabled( )
cmd     YDisplay target get_enabled
```

Returns :

either Y_ENABLED_FALSE or Y_ENABLED_TRUE, according to true if the screen is powered, false otherwise

On failure, throws an exception or returns Y_ENABLED_INVALID.

display→get_errorMessage() YDisplay
display→errorMessage()

Returns the error message of the latest error with the display.

```
js      function get_errorMessage( )
nodejs  function get_errorMessage( )
php     function get_errorMessage( )
cpp     string get_errorMessage( )
m       -(NSString*) errorMessage
pas     function get_errorMessage( ): string
vb      function get_errorMessage( ) As String
cs      string get_errorMessage( )
java    String get_errorMessage( )
py      def 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() YDisplay
display→errorType()

Returns the numerical error code of the latest error with the display.

```
js      function get_errorType( )
nodejs  function get_errorType( )
php     function get_errorType( )
cpp     YRETCODE get_errorType( )
pas     function get_errorType( ): YRETCODE
vb      function get_errorType( ) As YRETCODE
cs      YRETCODE get_errorType( )
java    int get_errorType( )
py      def 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 occured while using the display object

display→get_friendlyName() YDisplay
display→friendlyName()

Returns a global identifier of the display in the format MODULE_NAME.FUNCTION_NAME.

```
js      function get_friendlyName( )
nodejs  function get_friendlyName( )
php     function get_friendlyName( )
cpp     string get_friendlyName( )
m       -(NSString*) friendlyName
cs      string get_friendlyName( )
java    String get_friendlyName( )
py      def 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 exemple: MyCustomName.relay1)

Returns :

a string that uniquely identifies the display using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

display→get_functionDescriptor() YDisplay
display→functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

js	function **get_functionDescriptor**( )
nodejs	function **get_functionDescriptor**( )
php	function **get_functionDescriptor**( )
cpp	YFUN_DESCR **get_functionDescriptor**( )
m	-(YFUN_DESCR) functionDescriptor
pas	function **get_functionDescriptor**( ): YFUN_DESCR
vb	function **get_functionDescriptor**( ) As YFUN_DESCR
cs	YFUN_DESCR **get_functionDescriptor**( )
java	String **get_functionDescriptor**( )
py	def **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

display→get_functionId() YDisplay
display→functionId()

Returns the hardware identifier of the display, without reference to the module.

js	function **get_functionId**( )
nodejs	function **get_functionId**( )
php	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	def **get_functionId**( )

For example `relay1`

Returns :

a string that identifies the display (ex: `relay1`) On failure, throws an exception or returns `Y_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( )
nodejs  function get_hardwareId( )
php     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      def 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 Y_HARDWAREID_INVALID.

display→get_layerCount() YDisplay
display→layerCount()

Returns the number of available layers to draw on.

```
js      function get_layerCount( )
nodejs  function get_layerCount( )
php     function get_layerCount( )
cpp     int get_layerCount( )
m       -(int) layerCount
pas     function get_layerCount( ): LongInt
vb      function get_layerCount( ) As Integer
cs      int get_layerCount( )
java    int get_layerCount( )
py      def 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 `Y_LAYERCOUNT_INVALID`.

display→get_layerHeight() YDisplay
display→layerHeight()

Returns the height of the layers to draw on, in pixels.

```
js      function get_layerHeight( )
nodejs  function get_layerHeight( )
php     function get_layerHeight( )
cpp     int get_layerHeight( )
m       -(int) layerHeight
pas     function get_layerHeight( ): LongInt
vb      function get_layerHeight( ) As Integer
cs      int get_layerHeight( )
java    int get_layerHeight( )
py      def 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 `Y_LAYERHEIGHT_INVALID`.

display→get_layerWidth() YDisplay
display→layerWidth()

Returns the width of the layers to draw on, in pixels.

```
js      function get_layerWidth( )
nodejs  function get_layerWidth( )
php     function get_layerWidth( )
cpp     int get_layerWidth( )
m       -(int) layerWidth
pas     function get_layerWidth( ): LongInt
vb      function get_layerWidth( ) As Integer
cs      int get_layerWidth( )
java    int get_layerWidth( )
py      def 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 `Y_LAYERWIDTH_INVALID`.

display→get_logicalName() YDisplay
display→logicalName()

Returns the logical name of the display.

```
js      function get_logicalName( )
nodejs  function get_logicalName( )
php     function get_logicalName( )
cpp     string get_logicalName( )
m       -(NSString*) logicalName
pas     function get_logicalName( ): string
vb      function get_logicalName( ) As String
cs      string get_logicalName( )
java    String get_logicalName( )
py      def 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 `Y_LOGICALNAME_INVALID`.

display→get_module() YDisplay
display→module()

Gets the YModule object for the device on which the function is located.

```
js      function get_module( )
nodejs  function get_module( )
php     function get_module( )
cpp     YModule * get_module( )
m       -(YModule*) module
pas     function get_module( ): TYModule
vb      function get_module( ) As YModule
cs      YModule get_module( )
java    YModule get_module( )
py      def 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)
nodejs function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as online. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

display→get_orientation() YDisplay
display→orientation()

Returns the currently selected display orientation.

```
js      function get_orientation( )
nodejs  function get_orientation( )
php     function get_orientation( )
cpp     Y_ORIENTATION_enum get_orientation( )
m       -(Y_ORIENTATION_enum) orientation
pas     function get_orientation( ): Integer
vb      function get_orientation( ) As Integer
cs      int get_orientation( )
java    int get_orientation( )
py      def get_orientation( )
cmd     YDisplay target get_orientation
```

Returns :

a value among `Y_ORIENTATION_LEFT`, `Y_ORIENTATION_UP`, `Y_ORIENTATION_RIGHT` and `Y_ORIENTATION_DOWN` corresponding to the currently selected display orientation

On failure, throws an exception or returns `Y_ORIENTATION_INVALID`.

display→get_startupSeq() YDisplay
display→startupSeq()

Returns the name of the sequence to play when the displayed is powered on.

```
js      function get_startupSeq( )
nodejs  function get_startupSeq( )
php     function get_startupSeq( )
cpp     string get_startupSeq( )
m       -(NSString*) startupSeq
pas     function get_startupSeq( ): string
vb      function get_startupSeq( ) As String
cs      string get_startupSeq( )
java    String get_startupSeq( )
py      def 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 `Y_STARTUPSEQ_INVALID`.

display→get_userData() YDisplay
display→userData()

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

js	function **get_userData**( )
nodejs	function **get_userData**( )
php	function **get_userData**( )
cpp	void * **get_userData**( )
m	-(void*) userData
pas	function **get_userData**( ): Tobject
vb	function **get_userData**( ) As Object
cs	object **get_userData**( )
java	Object **get_userData**( )
py	def **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( )
nodejs  function isOnline( )
php     function isOnline( )
cpp     bool isOnline( )
m       -(BOOL) isOnline
pas     function isOnline( ): boolean
vb      function isOnline( ) As Boolean
cs      bool isOnline( )
java    boolean isOnline( )
py      def 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)
nodejs  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→load() YDisplay

Preloads the display cache with a specified validity duration.

```
js      function load( msValidity)
nodejs  function load( msValidity)
php     function load( $msValidity)
cpp     YRETCODE load( int msValidity)
m       -(YRETCODE) load : (int) msValidity
pas     function load( msValidity: integer): YRETCODE
vb      function load( ByVal msValidity As Integer) As YRETCODE
cs      YRETCODE load( int msValidity)
java    int load( long msValidity)
py      def 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 trafic 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→load_async() YDisplay

Preloads the display cache with a specified validity duration (asynchronous version).

```
js      function load_async( msValidity, callback, context)
nodejs  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 trafic 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→newSequence() YDisplay

Starts to record all display commands into a sequence, for later replay.

```
js      function newSequence( )
nodejs  function newSequence( )
php     function newSequence( )
cpp     int newSequence( )
m       -(int) newSequence
pas     function newSequence( ): LongInt
vb      function newSequence( ) As Integer
cs      int newSequence( )
java    int newSequence( )
py      def 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( )
nodejs  function nextDisplay( )
php     function nextDisplay( )
cpp     YDisplay * nextDisplay( )
m       -(YDisplay*) nextDisplay
pas     function nextDisplay( ): TYDisplay
vb      function nextDisplay( ) As YDisplay
cs      YDisplay nextDisplay( )
java    YDisplay nextDisplay( )
py      def nextDisplay( )
```

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)
nodejs  function pauseSequence( delay_ms)
php     function pauseSequence( $delay_ms)
cpp     int pauseSequence( int delay_ms)
m       -(int) pauseSequence : (int) delay_ms
pas     function pauseSequence( delay_ms: LongInt): LongInt
vb      function pauseSequence( ) As Integer
cs      int pauseSequence( int delay_ms)
java    int pauseSequence( int delay_ms)
py      def pauseSequence( 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)
nodejs  function playSequence( sequenceName)
php     function playSequence( $sequenceName)
cpp     int playSequence( string sequenceName)
m       -(int) playSequence : (NSString*) sequenceName
pas     function playSequence( sequenceName: string): LongInt
vb      function playSequence( ) As Integer
cs      int playSequence( string sequenceName)
java    int playSequence( String sequenceName)
py      def playSequence( 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)
nodejs  function registerValueCallback( callback)
php     function registerValueCallback( $callback)
cpp     int registerValueCallback( YDisplayValueCallback callback)
m       -(int) registerValueCallback : (YDisplayValueCallback) callback
pas     function registerValueCallback( callback: TYDisplayValueCallback): LongInt
vb      function registerValueCallback( ) As Integer
cs      int registerValueCallback( ValueCallback callback)
java    int registerValueCallback( UpdateCallback callback)
py      def 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( )
nodejs  function resetAll( )
php     function resetAll( )
cpp     int resetAll( )
m       -(int) resetAll
pas     function resetAll( ): LongInt
vb      function resetAll( ) As Integer
cs      int resetAll( )
java    int resetAll( )
py      def resetAll( )
cmd     YDisplay target resetAll
```

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)
nodejs  function saveSequence( sequenceName)
php     function saveSequence( $sequenceName)
cpp     int saveSequence( string sequenceName)
m       -(int) saveSequence : (NSString*) sequenceName
pas     function saveSequence( sequenceName: string): LongInt
vb      function saveSequence( ) As Integer
cs      int saveSequence( string sequenceName)
java    int saveSequence( String sequenceName)
py      def saveSequence( 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() YDisplay
display→setBrightness()

Changes the brightness of the display.

```
js      function set_brightness( newval)
nodejs  function set_brightness( newval)
php     function set_brightness( $newval)
cpp     int set_brightness( int newval)
m       -(int) setBrightness : (int) newval
pas     function 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)
py      def set_brightness( 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)
nodejs  function set_enabled( newval)
php     function set_enabled( $newval)
cpp     int set_enabled( Y_ENABLED_enum newval)
m       -(int) setEnabled : (Y_ENABLED_enum) newval
pas     function 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)
py      def set_enabled( newval)
cmd     YDisplay target set_enabled newval
```

Parameters :

newval either Y_ENABLED_FALSE or Y_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() YDisplay
display→setLogicalName()

Changes the logical name of the display.

```
js      function set_logicalName( newval)
nodejs  function set_logicalName( newval)
php     function set_logicalName( $newval)
cpp     int set_logicalName( const string& newval)
m       -(int) setLogicalName : (NSString*) newval
pas     function 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)
py      def set_logicalName( 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)
nodejs  function set_orientation( newval)
php     function set_orientation( $newval)
cpp     int set_orientation( Y_ORIENTATION_enum newval)
m       -(int) setOrientation : (Y_ORIENTATION_enum) newval
pas     function 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)
py      def set_orientation( 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 `Y_ORIENTATION_LEFT`, `Y_ORIENTATION_UP`, `Y_ORIENTATION_RIGHT` and `Y_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**)
nodejs	function **set_startupSeq**(**newval**)
php	function **set_startupSeq**(**$newval**)
cpp	int **set_startupSeq**(const string& **newval**)
m	-(int) setStartupSeq : (NSString*) **newval**
pas	function **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**)
py	def **set_startupSeq**(**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)
nodejs  function set_userData( data)
php     function set_userData( $data)
cpp     void set_userData( void* data)
m       -(void) setUserData : (void*) data
pas     procedure 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      def 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( )
nodejs  function stopSequence( )
php     function stopSequence( )
cpp     int stopSequence( )
m       -(int) stopSequence
pas     function stopSequence( ): LongInt
vb      function stopSequence( ) As Integer
cs      int stopSequence( )
java    int stopSequence( )
py      def 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)
nodejs  function swapLayerContent( layerIdA, layerIdB)
php     function swapLayerContent( $layerIdA, $layerIdB)
cpp     int swapLayerContent( int layerIdA, int layerIdB)
m       -(int) swapLayerContent : (int) layerIdA
                                : (int) layerIdB
pas     function swapLayerContent( layerIdA: LongInt, layerIdB: LongInt): LongInt
vb      function swapLayerContent( ) As Integer
cs      int swapLayerContent( int layerIdA, int layerIdB)
java    int swapLayerContent( int layerIdA, int layerIdB)
py      def swapLayerContent( layerIdA, 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 onae 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→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)
nodejs  function upload( pathname, content)
php     function upload( $pathname, $content)
cpp     int upload( string pathname, string content)
m       -(int) upload : (NSString*) pathname
                      : (NSData*) content
pas     function upload( pathname: string, content: TByteArray): LongInt
vb      procedure upload( )
cs      int upload( string pathname)
java    int upload( String pathname)
py      def upload( pathname, 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)
nodejs  function 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.

21.4. DisplayLayer object interface

A DisplayLayer is 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>
nodejs  var yoctolib = require('yoctolib');
        var YDisplay = yoctolib.YDisplay;
php     require_once('yocto_display.php');
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;
py      from yocto_display import *
```

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 text drawing and 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 behaviour 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( )
nodejs  function clear( )
php     function clear( )
cpp     int clear( )
m       -(int) clear
pas     function clear( ): LongInt
vb      function clear( ) As Integer
cs      int clear( )
java    int clear( )
py      def 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( )
nodejs  function clearConsole( )
php     function clearConsole( )
cpp     int clearConsole( )
m       -(int) clearConsole
pas     function clearConsole( ): LongInt
vb      function clearConsole( ) As Integer
cs      int clearConsole( )
java    int clearConsole( )
py      def 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)
nodejs  function consoleOut( text)
php     function consoleOut( $text)
cpp     int consoleOut( string text)
m       -(int) consoleOut : (NSString*) text
pas     function consoleOut( text: string): LongInt
vb      function consoleOut( ) As Integer
cs      int consoleOut( string text)
java    int consoleOut( String text)
py      def consoleOut( 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)
nodejs  function drawBar( x1, y1, x2, y2)
php     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     function drawBar( x1: LongInt,
                          y1: LongInt,
                          x2: LongInt,
                          y2: LongInt): LongInt
vb      function drawBar( ) As Integer
cs      int drawBar( int x1, int y1, int x2, int y2)
java    int drawBar( int x1, int y1, int x2, int y2)
py      def drawBar( x1, y1, x2, 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)
nodejs  function drawBitmap( x, y, w, bitmap, bgcol)
php     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     function drawBitmap( x: LongInt,
                             y: LongInt,
                             w: LongInt,
                             bitmap: TByteArray,
                             bgcol: LongInt): LongInt
vb      procedure drawBitmap( )
cs      int drawBitmap( int x,
                        int y,
                        int w,
                        int bgcol)
java    int drawBitmap( int x, int y, int w, int bgcol)
py      def drawBitmap( x, y, w, bitmap, 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)
nodejs  function drawCircle( x, y, r)
php     function drawCircle( $x, $y, $r)
cpp     int drawCircle( int x, int y, int r)
m       -(int) drawCircle : (int) x
                          : (int) y
                          : (int) r
pas     function drawCircle( x: LongInt, y: LongInt, r: LongInt): LongInt
vb      function drawCircle( ) As Integer
cs      int drawCircle( int x, int y, int r)
java    int drawCircle( int x, int y, int r)
py      def drawCircle( x, y, 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)
nodejs  function drawDisc( x, y, r)
php     function drawDisc( $x, $y, $r)
cpp     int drawDisc( int x, int y, int r)
m       -(int) drawDisc : (int) x
                        : (int) y
                        : (int) r
pas     function drawDisc( x: LongInt, y: LongInt, r: LongInt): LongInt
vb      function drawDisc( ) As Integer
cs      int drawDisc( int x, int y, int r)
java    int drawDisc( int x, int y, int r)
py      def drawDisc( x, y, 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)
nodejs  function drawImage( x, y, imagename)
php     function drawImage( $x, $y, $imagename)
cpp     int drawImage( int x, int y, string imagename)
m       -(int) drawImage : (int) x
                         : (int) y
                         : (NSString*) imagename
pas     function drawImage( x: LongInt, y: LongInt, imagename: string): LongInt
vb      function drawImage( ) As Integer
cs      int drawImage( int x, int y, string imagename)
java    int drawImage( int x, int y, String imagename)
py      def drawImage( x, y, 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)
nodejs  function drawPixel( x, y)
php     function drawPixel( $x, $y)
cpp     int drawPixel( int x, int y)
m       -(int) drawPixel : (int) x
                         : (int) y
pas     function drawPixel( x: LongInt, y: LongInt): LongInt
vb      function drawPixel( ) As Integer
cs      int drawPixel( int x, int y)
java    int drawPixel( int x, int y)
py      def drawPixel( x, 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)
nodejs  function drawRect( x1, y1, x2, y2)
php     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     function drawRect( x1: LongInt,
                           y1: LongInt,
                           x2: LongInt,
                           y2: LongInt): LongInt
vb      function drawRect( ) As Integer
cs      int drawRect( int x1, int y1, int x2, int y2)
java    int drawRect( int x1, int y1, int x2, int y2)
py      def drawRect( x1, y1, x2, 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)
nodejs  function drawText( x, y, anchor, text)
php     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     function drawText( x: LongInt,
                           y: LongInt,
                           anchor: TYALIGN,
                           text: string): LongInt
vb      function drawText( ) As Integer
cs      int drawText( int x, int y, ALIGN anchor, string text)
java    int drawText( int x, int y, ALIGN anchor, String text)
py      def drawText( x, y, anchor, 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 ancor point, in pixels
y	the distance from top of layer to the text ancor point, in pixels
anchor	the text anchor point, chosen among the Y_ALIGN enumeration: Y_ALIGN_TOP_LEFT, Y_ALIGN_CENTER_LEFT, Y_ALIGN_BASELINE_LEFT, Y_ALIGN_BOTTOM_LEFT, Y_ALIGN_TOP_CENTER, Y_ALIGN_CENTER, Y_ALIGN_BASELINE_CENTER, Y_ALIGN_BOTTOM_CENTER, Y_ALIGN_TOP_DECIMAL, Y_ALIGN_CENTER_DECIMAL, Y_ALIGN_BASELINE_DECIMAL, Y_ALIGN_BOTTOM_DECIMAL, Y_ALIGN_TOP_RIGHT, Y_ALIGN_CENTER_RIGHT, Y_ALIGN_BASELINE_RIGHT, Y_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() YDisplayLayer
displaylayer→display()

Gets parent YDisplay.

```
js      function get_display( )
nodejs  function get_display( )
php     function get_display( )
cpp     YDisplay* get_display( )
m       -(YDisplay*) display
pas     function get_display( ): TYDisplay
vb      function get_display( ) As YDisplay
cs      YDisplay get_display( )
java    YDisplay get_display( )
py      def get_display( )
```

Returns the parent YDisplay object of the current YDisplayLayer.

Returns :
an `YDisplay` object

displaylayer→get_displayHeight() YDisplayLayer
displaylayer→displayHeight()

Returns the display height, in pixels.

js	function **get_displayHeight**( )
nodejs	function **get_displayHeight**( )
php	function **get_displayHeight**( )
cpp	int **get_displayHeight**( )
m	-(int) displayHeight
pas	function **get_displayHeight**( ): LongInt
vb	function **get_displayHeight**( ) As Integer
cs	int **get_displayHeight**( )
java	int **get_displayHeight**( )
py	def **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() YDisplayLayer
displaylayer→displayWidth()

Returns the display width, in pixels.

js	function **get_displayWidth**( )
nodejs	function **get_displayWidth**( )
php	function **get_displayWidth**( )
cpp	int **get_displayWidth**( )
m	-(int) displayWidth
pas	function **get_displayWidth**( ): LongInt
vb	function **get_displayWidth**( ) As Integer
cs	int **get_displayWidth**( )
java	int **get_displayWidth**( )
py	def **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**( )
nodejs	function **get_layerHeight**( )
php	function **get_layerHeight**( )
cpp	int **get_layerHeight**( )
m	-(int) layerHeight
pas	function **get_layerHeight**( ): LongInt
vb	function **get_layerHeight**( ) As Integer
cs	int **get_layerHeight**( )
java	int **get_layerHeight**( )
py	def **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() YDisplayLayer
displaylayer→layerWidth()

Returns the width of the layers to draw on, in pixels.

```
js      function get_layerWidth( )
nodejs  function get_layerWidth( )
php     function get_layerWidth( )
cpp     int get_layerWidth( )
m       -(int) layerWidth
pas     function get_layerWidth( ): LongInt
vb      function get_layerWidth( ) As Integer
cs      int get_layerWidth( )
java    int get_layerWidth( )
py      def 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**( )
nodejs	function **hide**( )
php	function **hide**( )
cpp	int **hide**( )
m	-(int) **hide**
pas	function **hide**( ): LongInt
vb	function **hide**( ) As Integer
cs	int **hide**( )
java	int **hide**( )
py	def **hide**( )
cmd	YDisplay **target** [-layer **layerId**] **hide**

The state of the layer is perserved 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)
nodejs  function lineTo( x, y)
php     function lineTo( $x, $y)
cpp     int lineTo( int x, int y)
m       -(int) lineTo : (int) x
                      : (int) y
pas     function lineTo( x: LongInt, y: LongInt): LongInt
vb      function lineTo( ) As Integer
cs      int lineTo( int x, int y)
java    int lineTo( int x, int y)
py      def lineTo( x, 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)
nodejs  function moveTo( x, y)
php     function moveTo( $x, $y)
cpp     int moveTo( int x, int y)
m       -(int) moveTo : (int) x
                      : (int) y
pas     function moveTo( x: LongInt, y: LongInt): LongInt
vb      function moveTo( ) As Integer
cs      int moveTo( int x, int y)
java    int moveTo( int x, int y)
py      def moveTo( x, 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( )
nodejs  function reset( )
php     function reset( )
cpp     int reset( )
m       -(int) reset
pas     function reset( ): LongInt
vb      function reset( ) As Integer
cs      int reset( )
java    int reset( )
py      def 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)
nodejs  function selectColorPen( color)
php     function selectColorPen( $color)
cpp     int selectColorPen( int color)
m       -(int) selectColorPen : (int) color
pas     function selectColorPen( color: LongInt): LongInt
vb      function selectColorPen( ) As Integer
cs      int selectColorPen( int color)
java    int selectColorPen( int color)
py      def selectColorPen( 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 text drawing and bitmap copy functions.

```
js      function selectEraser( )
nodejs  function selectEraser( )
php     function selectEraser( )
cpp     int selectEraser( )
m       -(int) selectEraser
pas     function selectEraser( ): LongInt
vb      function selectEraser( ) As Integer
cs      int selectEraser( )
java    int selectEraser( )
py      def 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)
nodejs  function selectFont( fontname)
php     function selectFont( $fontname)
cpp     int selectFont( string fontname)
m       -(int) selectFont : (NSString*) fontname
pas     function selectFont( fontname: string): LongInt
vb      function selectFont( ) As Integer
cs      int selectFont( string fontname)
java    int selectFont( String fontname)
py      def selectFont( 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

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)
nodejs  function selectGrayPen( graylevel)
php     function selectGrayPen( $graylevel)
cpp     int selectGrayPen( int graylevel)
m       -(int) selectGrayPen : (int) graylevel
pas     function selectGrayPen( graylevel: LongInt): LongInt
vb      function selectGrayPen( ) As Integer
cs      int selectGrayPen( int graylevel)
java    int selectGrayPen( int graylevel)
py      def selectGrayPen( 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 lighest 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)
nodejs  function setAntialiasingMode( mode)
php     function setAntialiasingMode( $mode)
cpp     int setAntialiasingMode( bool mode)
m       -(int) setAntialiasingMode : (bool) mode
pas     function setAntialiasingMode( mode: boolean): LongInt
vb      function setAntialiasingMode( ) As Integer
cs      int setAntialiasingMode( bool mode)
java    int setAntialiasingMode( boolean mode)
py      def setAntialiasingMode( mode)
cmd     YDisplay target [-layer layerId] setAntialiasingMode  mode
```

Anti-aliasing provides a smoother aspect when looked from far enough, but it can add fuzzyness 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 antialiasing, 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)
nodejs  function setConsoleBackground( bgcol)
php     function setConsoleBackground( $bgcol)
cpp     int setConsoleBackground( int bgcol)
m       -(int) setConsoleBackground : (int) bgcol
pas     function setConsoleBackground( bgcol: LongInt): LongInt
vb      function setConsoleBackground( ) As Integer
cs      int setConsoleBackground( int bgcol)
java    int setConsoleBackground( int bgcol)
py      def setConsoleBackground( 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)
nodejs  function setConsoleMargins( x1, y1, x2, y2)
php     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     function setConsoleMargins( x1: LongInt,
                                    y1: LongInt,
                                    x2: LongInt,
                                    y2: LongInt): LongInt
vb      function setConsoleMargins( ) As Integer
cs      int setConsoleMargins( int x1, int y1, int x2, int y2)
java    int setConsoleMargins( int x1, int y1, int x2, int y2)
py      def setConsoleMargins( x1, y1, x2, 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 behaviour used by the `consoleOut` function.

```
js      function setConsoleWordWrap( wordwrap)
nodejs  function setConsoleWordWrap( wordwrap)
php     function setConsoleWordWrap( $wordwrap)
cpp     int setConsoleWordWrap( bool wordwrap)
m       -(int) setConsoleWordWrap : (bool) wordwrap
pas     function setConsoleWordWrap( wordwrap: boolean): LongInt
vb      function setConsoleWordWrap( ) As Integer
cs      int setConsoleWordWrap( bool wordwrap)
java    int setConsoleWordWrap( boolean wordwrap)
py      def setConsoleWordWrap( 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)
nodejs  function setLayerPosition( x, y, scrollTime)
php     function setLayerPosition( $x, $y, $scrollTime)
cpp     int setLayerPosition( int x, int y, int scrollTime)
m       -(int) setLayerPosition : (int) x
                                : (int) y
                                : (int) scrollTime
pas     function setLayerPosition( x: LongInt,
                                   y: LongInt,
                                   scrollTime: LongInt): LongInt
vb      function setLayerPosition( ) As Integer
cs      int setLayerPosition( int x, int y, int scrollTime)
java    int setLayerPosition( int x, int y, int scrollTime)
py      def setLayerPosition( x, y, 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( )
nodejs  function unhide( )
php     function unhide( )
cpp     int unhide( )
m       -(int) unhide
pas     function unhide( ): LongInt
vb      function unhide( ) As Integer
cs      int unhide( )
java    int unhide( )
py      def 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.

21.5. AnButton function interface

Yoctopuce application programming interface allows you 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:

```
js      <script type='text/javascript' src='yocto_anbutton.js'></script>
nodejs  var yoctolib = require('yoctolib');
        var YAnButton = yoctolib.YAnButton;
php     require_once('yocto_anbutton.php');
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;
py      from yocto_anbutton import *
```

Global functions
yFindAnButton(**func**) Retrieves an analog input for a given identifier.
yFirstAnButton() Starts the enumeration of analog inputs currently accessible.
YAnButton methods
anbutton→describe() Returns a short text that describes 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_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 transitionned 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 transitionned 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_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→load(**msValidity**)

Preloads the analog input cache with a specified validity duration.

anbutton→load_async(**msValidity**, **callback**, **context**)

Preloads the analog input cache with a specified validity duration (asynchronous version).

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 his 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_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→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 yFindAnButton()

Retrieves an analog input for a given identifier.

```
js      function yFindAnButton( func)
nodejs  function FindAnButton( func)
php     function yFindAnButton( $func)
cpp     YAnButton* yFindAnButton( const string& func)
m       YAnButton* yFindAnButton( NSString* func)
pas     function yFindAnButton( func: string): TYAnButton
vb      function yFindAnButton( ByVal func As String) As YAnButton
cs      YAnButton FindAnButton( string func)
java    YAnButton FindAnButton( String func)
py      def FindAnButton( 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 :

func a string that uniquely characterizes the analog input

Returns :

a `YAnButton` object allowing you to drive the analog input.

YAnButton.FirstAnButton() YAnButton yFirstAnButton()

Starts the enumeration of analog inputs currently accessible.

```
js      function yFirstAnButton( )
nodejs  function FirstAnButton( )
php     function yFirstAnButton( )
cpp     YAnButton* yFirstAnButton( )
m       YAnButton* yFirstAnButton( )
pas     function yFirstAnButton( ): TYAnButton
vb      function yFirstAnButton( ) As YAnButton
cs      YAnButton FirstAnButton( )
java    YAnButton FirstAnButton( )
py      def 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.

anbutton→describe() YAnButton

Returns a short text that describes the analog input in the form TYPE(NAME)=SERIAL.FUNCTIONID.

js	function **describe**( )
nodejs	function **describe**( )
php	function **describe**( )
cpp	string **describe**( )
m	-(NSString*) **describe**
pas	function **describe**( ): string
vb	function **describe**( ) As String
cs	string **describe**( )
java	String **describe**( )
py	def **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() YAnButton
anbutton→advertisedValue()

Returns the current value of the analog input (no more than 6 characters).

```
js      function get_advertisedValue( )
nodejs  function get_advertisedValue( )
php     function get_advertisedValue( )
cpp     string get_advertisedValue( )
m       -(NSString*) advertisedValue
pas     function get_advertisedValue( ): string
vb      function get_advertisedValue( ) As String
cs      string get_advertisedValue( )
java    String get_advertisedValue( )
py      def 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 `Y_ADVERTISEDVALUE_INVALID`.

anbutton→get_analogCalibration() YAnButton
anbutton→analogCalibration()

Tells if a calibration process is currently ongoing.

```
js      function get_analogCalibration( )
nodejs  function get_analogCalibration( )
php     function get_analogCalibration( )
cpp     Y_ANALOGCALIBRATION_enum get_analogCalibration( )
m       -(Y_ANALOGCALIBRATION_enum) analogCalibration
pas     function get_analogCalibration( ): Integer
vb      function get_analogCalibration( ) As Integer
cs      int get_analogCalibration( )
java    int get_analogCalibration( )
py      def get_analogCalibration( )
cmd     YAnButton target get_analogCalibration
```

Returns :

either `Y_ANALOGCALIBRATION_OFF` or `Y_ANALOGCALIBRATION_ON`

On failure, throws an exception or returns `Y_ANALOGCALIBRATION_INVALID`.

anbutton→get_calibratedValue() YAnButton
anbutton→calibratedValue()

Returns the current calibrated input value (between 0 and 1000, included).

```
js      function get_calibratedValue( )
nodejs  function get_calibratedValue( )
php     function get_calibratedValue( )
cpp     int get_calibratedValue( )
m       -(int) calibratedValue
pas     function get_calibratedValue( ): LongInt
vb      function get_calibratedValue( ) As Integer
cs      int get_calibratedValue( )
java    int get_calibratedValue( )
py      def 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 `Y_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**( )
nodejs	function **get_calibrationMax**( )
php	function **get_calibrationMax**( )
cpp	int **get_calibrationMax**( )
m	-(int) calibrationMax
pas	function **get_calibrationMax**( ): LongInt
vb	function **get_calibrationMax**( ) As Integer
cs	int **get_calibrationMax**( )
java	int **get_calibrationMax**( )
py	def **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 `Y_CALIBRATIONMAX_INVALID`.

anbutton→get_calibrationMin() YAnButton
anbutton→calibrationMin()

Returns the minimal value measured during the calibration (between 0 and 4095, included).

```
js      function get_calibrationMin( )
nodejs  function get_calibrationMin( )
php     function get_calibrationMin( )
cpp     int get_calibrationMin( )
m       -(int) calibrationMin
pas     function get_calibrationMin( ): LongInt
vb      function get_calibrationMin( ) As Integer
cs      int get_calibrationMin( )
java    int get_calibrationMin( )
py      def 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 `Y_CALIBRATIONMIN_INVALID`.

anbutton→get_errorMessage() YAnButton
anbutton→errorMessage()

Returns the error message of the latest error with the analog input.

```
js      function get_errorMessage( )
nodejs  function get_errorMessage( )
php     function get_errorMessage( )
cpp     string get_errorMessage( )
m       -(NSString*) errorMessage
pas     function get_errorMessage( ): string
vb      function get_errorMessage( ) As String
cs      string get_errorMessage( )
java    String get_errorMessage( )
py      def 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() YAnButton
anbutton→errorType()

Returns the numerical error code of the latest error with the analog input.

```
js      function get_errorType( )
nodejs  function get_errorType( )
php     function get_errorType( )
cpp     YRETCODE get_errorType( )
pas     function get_errorType( ): YRETCODE
vb      function get_errorType( ) As YRETCODE
cs      YRETCODE get_errorType( )
java    int get_errorType( )
py      def 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 occured while using the analog input object

anbutton→get_friendlyName() YAnButton
anbutton→friendlyName()

Returns a global identifier of the analog input in the format MODULE_NAME.FUNCTION_NAME.

```
js      function get_friendlyName( )
nodejs  function get_friendlyName( )
php     function get_friendlyName( )
cpp     string get_friendlyName( )
m       -(NSString*) friendlyName
cs      string get_friendlyName( )
java    String get_friendlyName( )
py      def 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 exemple: MyCustomName.relay1)

Returns :

a string that uniquely identifies the analog input using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

anbutton→get_functionDescriptor() YAnButton
anbutton→functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

js	function **get_functionDescriptor**( )
nodejs	function **get_functionDescriptor**( )
php	function **get_functionDescriptor**( )
cpp	YFUN_DESCR **get_functionDescriptor**( )
m	-(YFUN_DESCR) functionDescriptor
pas	function **get_functionDescriptor**( ): YFUN_DESCR
vb	function **get_functionDescriptor**( ) As YFUN_DESCR
cs	YFUN_DESCR **get_functionDescriptor**( )
java	String **get_functionDescriptor**( )
py	def **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

anbutton→get_functionId() YAnButton
anbutton→functionId()

Returns the hardware identifier of the analog input, without reference to the module.

```
js      function get_functionId( )
nodejs  function get_functionId( )
php     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      def get_functionId( )
```

For example `relay1`

Returns :

a string that identifies the analog input (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

anbutton→get_hardwareId() YAnButton
anbutton→hardwareId()

Returns the unique hardware identifier of the analog input in the form SERIAL.FUNCTIONID.

js	function **get_hardwareId**( )
nodejs	function **get_hardwareId**( )
php	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	def **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 Y_HARDWAREID_INVALID.

anbutton→get_isPressed() YAnButton
anbutton→isPressed()

Returns true if the input (considered as binary) is active (closed contact), and false otherwise.

```
js      function get_isPressed( )
nodejs  function get_isPressed( )
php     function get_isPressed( )
cpp     Y_ISPRESSED_enum get_isPressed( )
m       -(Y_ISPRESSED_enum) isPressed
pas     function get_isPressed( ): Integer
vb      function get_isPressed( ) As Integer
cs      int get_isPressed( )
java    int get_isPressed( )
py      def get_isPressed( )
cmd     YAnButton target get_isPressed
```

Returns :

either `Y_ISPRESSED_FALSE` or `Y_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 `Y_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 transitionned from open to closed).

```
js      function get_lastTimePressed( )
nodejs  function get_lastTimePressed( )
php     function get_lastTimePressed( )
cpp     s64 get_lastTimePressed( )
m       -(s64) lastTimePressed
pas     function get_lastTimePressed( ): int64
vb      function get_lastTimePressed( ) As Long
cs      long get_lastTimePressed( )
java    long get_lastTimePressed( )
py      def 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 transitionned from open to closed)

On failure, throws an exception or returns `Y_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 transitionned from closed to open).

```
js      function get_lastTimeReleased( )
nodejs  function get_lastTimeReleased( )
php     function get_lastTimeReleased( )
cpp     s64 get_lastTimeReleased( )
m       -(s64) lastTimeReleased
pas     function get_lastTimeReleased( ): int64
vb      function get_lastTimeReleased( ) As Long
cs      long get_lastTimeReleased( )
java    long get_lastTimeReleased( )
py      def 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 transitionned from closed to open)

On failure, throws an exception or returns `Y_LASTTIMERELEASED_INVALID`.

anbutton→get_logicalName() YAnButton
anbutton→logicalName()

Returns the logical name of the analog input.

```
js      function get_logicalName( )
nodejs  function get_logicalName( )
php     function get_logicalName( )
cpp     string get_logicalName( )
m       -(NSString*) logicalName
pas     function get_logicalName( ): string
vb      function get_logicalName( ) As String
cs      string get_logicalName( )
java    String get_logicalName( )
py      def 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 `Y_LOGICALNAME_INVALID`.

anbutton→get_module() YAnButton
anbutton→module()

Gets the YModule object for the device on which the function is located.

```
js      function get_module( )
nodejs  function get_module( )
php     function get_module( )
cpp     YModule * get_module( )
m       -(YModule*) module
pas     function get_module( ): TYModule
vb      function get_module( ) As YModule
cs      YModule get_module( )
java    YModule get_module( )
py      def 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() YAnButton
anbutton→module_async()

Gets the YModule object for the device on which the function is located (asynchronous version).

```
js      function get_module_async( callback, context)
nodejs  function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as online. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

anbutton→get_pulseCounter() YAnButton
anbutton→pulseCounter()

Returns the pulse counter value

js	function **get_pulseCounter**( )
nodejs	function **get_pulseCounter**( )
php	function **get_pulseCounter**( )
cpp	s64 **get_pulseCounter**( )
m	-(s64) pulseCounter
pas	function **get_pulseCounter**( ): int64
vb	function **get_pulseCounter**( ) As Long
cs	long **get_pulseCounter**( )
java	long **get_pulseCounter**( )
py	def **get_pulseCounter**( )

Returns :
an integer corresponding to the pulse counter value

On failure, throws an exception or returns `Y_PULSECOUNTER_INVALID`.

anbutton→get_pulseTimer() YAnButton
anbutton→pulseTimer()

Returns the timer of the pulses counter (ms)

```
js      function get_pulseTimer( )
nodejs  function get_pulseTimer( )
php     function get_pulseTimer( )
cpp     s64 get_pulseTimer( )
m       -(s64) pulseTimer
pas     function get_pulseTimer( ): int64
vb      function get_pulseTimer( ) As Long
cs      long get_pulseTimer( )
java    long get_pulseTimer( )
py      def get_pulseTimer( )
```

Returns :

an integer corresponding to the timer of the pulses counter (ms)

On failure, throws an exception or returns `Y_PULSETIMER_INVALID`.

anbutton→get_rawValue() YAnButton
anbutton→rawValue()

Returns the current measured input value as-is (between 0 and 4095, included).

```
js      function get_rawValue( )
nodejs  function get_rawValue( )
php     function get_rawValue( )
cpp     int get_rawValue( )
m       -(int) rawValue
pas     function get_rawValue( ): LongInt
vb      function get_rawValue( ) As Integer
cs      int get_rawValue( )
java    int get_rawValue( )
py      def 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 `Y_RAWVALUE_INVALID`.

anbutton→get_sensitivity() YAnButton
anbutton→sensitivity()

Returns the sensibility for the input (between 1 and 1000) for triggering user callbacks.

```
js      function get_sensitivity( )
nodejs  function get_sensitivity( )
php     function get_sensitivity( )
cpp     int get_sensitivity( )
m       -(int) sensitivity
pas     function get_sensitivity( ): LongInt
vb      function get_sensitivity( ) As Integer
cs      int get_sensitivity( )
java    int get_sensitivity( )
py      def 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 `Y_SENSITIVITY_INVALID`.

anbutton→get_userData() YAnButton
anbutton→userData()

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

```
js      function get_userData( )
nodejs  function get_userData( )
php     function get_userData( )
cpp     void * get_userData( )
m       -(void*) userData
pas     function get_userData( ): Tobject
vb      function get_userData( ) As Object
cs      object get_userData( )
java    Object get_userData( )
py      def 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( )
nodejs  function isOnline( )
php     function isOnline( )
cpp     bool isOnline( )
m       -(BOOL) isOnline
pas     function isOnline( ): boolean
vb      function isOnline( ) As Boolean
cs      bool isOnline( )
java    boolean isOnline( )
py      def 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)
nodejs  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→load() YAnButton

Preloads the analog input cache with a specified validity duration.

```
js      function load( msValidity)
nodejs  function load( msValidity)
php     function load( $msValidity)
cpp     YRETCODE load( int msValidity)
m       -(YRETCODE) load : (int) msValidity
pas     function load( msValidity: integer): YRETCODE
vb      function load( ByVal msValidity As Integer) As YRETCODE
cs      YRETCODE load( int msValidity)
java    int load( long msValidity)
py      def 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 trafic 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→load_async() YAnButton

Preloads the analog input cache with a specified validity duration (asynchronous version).

```
js      function load_async( msValidity, callback, context)
nodejs  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 trafic 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→nextAnButton() YAnButton

Continues the enumeration of analog inputs started using `yFirstAnButton()`.

```
js      function nextAnButton( )
nodejs  function nextAnButton( )
php     function nextAnButton( )
cpp     YAnButton * nextAnButton( )
m       -(YAnButton*) nextAnButton
pas     function nextAnButton( ): TYAnButton
vb      function nextAnButton( ) As YAnButton
cs      YAnButton nextAnButton( )
java    YAnButton nextAnButton( )
py      def nextAnButton( )
```

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)
nodejs  function registerValueCallback( callback)
php     function registerValueCallback( $callback)
cpp     int registerValueCallback( YAnButtonValueCallback callback)
m       -(int) registerValueCallback : (YAnButtonValueCallback) callback
pas     function registerValueCallback( callback: TYAnButtonValueCallback): LongInt
vb      function registerValueCallback( ) As Integer
cs      int registerValueCallback( ValueCallback callback)
java    int registerValueCallback( UpdateCallback callback)
py      def 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 his timer

```
js      function resetCounter( )
nodejs  function resetCounter( )
php     function resetCounter( )
cpp     int resetCounter( )
m       -(int) resetCounter
pas     function resetCounter( ): integer
vb      function resetCounter( ) As Integer
cs      int resetCounter( )
java    int resetCounter( )
py      def 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)
nodejs  function set_analogCalibration( newval)
php     function set_analogCalibration( $newval)
cpp     int set_analogCalibration( Y_ANALOGCALIBRATION_enum newval)
m       -(int) setAnalogCalibration : (Y_ANALOGCALIBRATION_enum) newval
pas     function 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)
py      def set_analogCalibration( 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 `Y_ANALOGCALIBRATION_OFF` or `Y_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)
nodejs  function set_calibrationMax( newval)
php     function set_calibrationMax( $newval)
cpp     int set_calibrationMax( int newval)
m       -(int) setCalibrationMax : (int) newval
pas     function 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)
py      def set_calibrationMax( 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() YAnButton
anbutton→setCalibrationMin()

Changes the minimal calibration value for the input (between 0 and 4095, included), without actually starting the automated calibration.

```
js      function set_calibrationMin( newval)
nodejs  function set_calibrationMin( newval)
php     function set_calibrationMin( $newval)
cpp     int set_calibrationMin( int newval)
m       -(int) setCalibrationMin : (int) newval
pas     function 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)
py      def set_calibrationMin( 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_logicalName() YAnButton
anbutton→setLogicalName()

Changes the logical name of the analog input.

```
js      function set_logicalName( newval)
nodejs  function set_logicalName( newval)
php     function set_logicalName( $newval)
cpp     int set_logicalName( const string& newval)
m       -(int) setLogicalName : (NSString*) newval
pas     function 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)
py      def set_logicalName( 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)
nodejs  function set_sensitivity( newval)
php     function set_sensitivity( $newval)
cpp     int set_sensitivity( int newval)
m       -(int) setSensitivity : (int) newval
pas     function 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)
py      def set_sensitivity( 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() YAnButton
anbutton→setUserData()

Stores a user context provided as argument in the userData attribute of the function.

```
js      function set_userData( data)
nodejs  function set_userData( data)
php     function set_userData( $data)
cpp     void set_userData( void* data)
m       -(void) setUserData : (void*) data
pas     procedure 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      def 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→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)
nodejs  function 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.

21.6. Files function interface

The filesystem interface makes it possible to store files on some devices, 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>
nodejs  var yoctolib = require('yoctolib');
        var YFiles = yoctolib.YFiles;
php     require_once('yocto_files.php');
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;
py      from yocto_files import *
```

Global functions
yFindFiles(**func**) Retrieves a filesystem for a given identifier.
yFirstFiles() Starts the enumeration of filesystems currently accessible.
YFiles methods
files→describe() Returns a short text that describes 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→format_fs() Reinitializes 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_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→load(**msValidity**)

Preloads the filesystem cache with a specified validity duration.

files→load_async(**msValidity**, **callback**, **context**)

Preloads the filesystem cache with a specified validity duration (asynchronous version).

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→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
yFindFiles()

Retrieves a filesystem for a given identifier.

```
js      function yFindFiles( func)
nodejs  function FindFiles( func)
php     function yFindFiles( $func)
cpp     YFiles* yFindFiles( string func)
m       +(YFiles*) yFindFiles : (NSString*) func
pas     function yFindFiles( func: string): TYFiles
vb      function yFindFiles( ByVal func As String) As YFiles
cs      YFiles FindFiles( string func)
java    YFiles FindFiles( String func)
py      def FindFiles( 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 :

func a string that uniquely characterizes the filesystem

Returns :

a `YFiles` object allowing you to drive the filesystem.

YFiles.FirstFiles() YFiles
yFirstFiles()

Starts the enumeration of filesystems currently accessible.

```
js      function yFirstFiles( )
nodejs  function FirstFiles( )
php     function yFirstFiles( )
cpp     YFiles* yFirstFiles( )
m       YFiles* yFirstFiles( )
pas     function yFirstFiles( ): TYFiles
vb      function yFirstFiles( ) As YFiles
cs      YFiles FirstFiles( )
java    YFiles FirstFiles( )
py      def 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.

files→describe() YFiles

Returns a short text that describes the filesystem in the form TYPE(NAME)=SERIAL.FUNCTIONID.

```
js      function describe( )
nodejs  function describe( )
php     function describe( )
cpp     string describe( )
m       -(NSString*) describe
pas     function describe( ): string
vb      function describe( ) As String
cs      string describe( )
java    String describe( )
py      def 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)
nodejs  function download( pathname)
php     function download( $pathname)
cpp     string download( string pathname)
m       -(NSData*) download : (NSString*) pathname
pas     function download( pathname: string): TByteArray
vb      function download( ) As Byte
py      def download( 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)
nodejs  function download_async( pathname, callback, context)
```

This is the asynchronous version that uses a callback to pass the result when the donwload 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→format_fs() YFiles

Reinitializes the filesystem to its clean, unfragmented, empty state.

```
js      function format_fs( )
nodejs  function format_fs( )
php     function format_fs( )
cpp     int format_fs( )
m       -(int) format_fs
pas     function format_fs( ): LongInt
vb      function format_fs( ) As Integer
cs      int format_fs( )
java    int format_fs( )
py      def format_fs( )
cmd     YFiles target format_fs
```

All files previously uploaded are permanently lost.

Returns :

YAPI_SUCCESS if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→get_advertisedValue() YFiles
files→advertisedValue()

Returns the current value of the filesystem (no more than 6 characters).

```
js      function get_advertisedValue( )
nodejs  function get_advertisedValue( )
php     function get_advertisedValue( )
cpp     string get_advertisedValue( )
m       -(NSString*) advertisedValue
pas     function get_advertisedValue( ): string
vb      function get_advertisedValue( ) As String
cs      string get_advertisedValue( )
java    String get_advertisedValue( )
py      def get_advertisedValue( )
cmd     YFiles target get_advertisedValue
```

Returns :

a string corresponding to the current value of the filesystem (no more than 6 characters). On failure, throws an exception or returns `Y_ADVERTISEDVALUE_INVALID`.

files→get_errorMessage() YFiles
files→errorMessage()

Returns the error message of the latest error with the filesystem.

```
js      function get_errorMessage( )
nodejs  function get_errorMessage( )
php     function get_errorMessage( )
cpp     string get_errorMessage( )
m       -(NSString*) errorMessage
pas     function get_errorMessage( ): string
vb      function get_errorMessage( ) As String
cs      string get_errorMessage( )
java    String get_errorMessage( )
py      def 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( )
nodejs  function get_errorType( )
php     function get_errorType( )
cpp     YRETCODE get_errorType( )
pas     function get_errorType( ): YRETCODE
vb      function get_errorType( ) As YRETCODE
cs      YRETCODE get_errorType( )
java    int get_errorType( )
py      def 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 occured while using the filesystem object

files→get_filesCount() YFiles
files→filesCount()

Returns the number of files currently loaded in the filesystem.

```
js      function get_filesCount( )
nodejs  function get_filesCount( )
php     function get_filesCount( )
cpp     int get_filesCount( )
m       -(int) filesCount
pas     function get_filesCount( ): LongInt
vb      function get_filesCount( ) As Integer
cs      int get_filesCount( )
java    int get_filesCount( )
py      def get_filesCount( )
cmd     YFiles target get_filesCount
```

Returns :
an integer corresponding to the number of files currently loaded in the filesystem

On failure, throws an exception or returns `Y_FILESCOUNT_INVALID`.

files→get_freeSpace() YFiles
files→freeSpace()

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

```
js      function get_freeSpace( )
nodejs  function get_freeSpace( )
php     function get_freeSpace( )
cpp     int get_freeSpace( )
m       -(int) freeSpace
pas     function get_freeSpace( ): LongInt
vb      function get_freeSpace( ) As Integer
cs      int get_freeSpace( )
java    int get_freeSpace( )
py      def get_freeSpace( )
cmd     YFiles target get_freeSpace
```

Returns :

an integer corresponding to the free space for uploading new files to the filesystem, in bytes

On failure, throws an exception or returns `Y_FREESPACE_INVALID`.

files→get_friendlyName() YFiles
files→friendlyName()

Returns a global identifier of the filesystem in the format MODULE_NAME.FUNCTION_NAME.

```
js      function get_friendlyName( )
nodejs  function get_friendlyName( )
php     function get_friendlyName( )
cpp     string get_friendlyName( )
m       -(NSString*) friendlyName
cs      string get_friendlyName( )
java    String get_friendlyName( )
py      def 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 exemple: MyCustomName.relay1)

Returns :

a string that uniquely identifies the filesystem using logical names (ex: MyCustomName.relay1) On failure, throws an exception or returns Y_FRIENDLYNAME_INVALID.

files→get_functionDescriptor() YFiles
files→functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

js	function **get_functionDescriptor**( )
nodejs	function **get_functionDescriptor**( )
php	function **get_functionDescriptor**( )
cpp	YFUN_DESCR **get_functionDescriptor**( )
m	-(YFUN_DESCR) functionDescriptor
pas	function **get_functionDescriptor**( ): YFUN_DESCR
vb	function **get_functionDescriptor**( ) As YFUN_DESCR
cs	YFUN_DESCR **get_functionDescriptor**( )
java	String **get_functionDescriptor**( )
py	def **get_functionDescriptor**( )

This identifier can be used to test if two instances of YFunction reference the same physical function on the same physical device.

Returns :

an identifier of type YFUN_DESCR. If the function has never been contacted, the returned value is Y_FUNCTIONDESCRIPTOR_INVALID.

files→get_functionId() YFiles
files→functionId()

Returns the hardware identifier of the filesystem, without reference to the module.

```
js      function get_functionId( )
nodejs  function get_functionId( )
php     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      def get_functionId( )
```

For example `relay1`

Returns :

a string that identifies the filesystem (ex: `relay1`) On failure, throws an exception or returns `Y_FUNCTIONID_INVALID`.

files→get_hardwareId() YFiles
files→hardwareId()

Returns the unique hardware identifier of the filesystem in the form SERIAL.FUNCTIONID.

```
js      function get_hardwareId( )
nodejs  function get_hardwareId( )
php     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      def get_hardwareId( )
```

The unique hardware identifier is composed of the device serial number and of the hardware identifier of the filesystem. (for example RELAYLO1-123456.relay1)

Returns :

a string that uniquely identifies the filesystem (ex: RELAYLO1-123456.relay1) On failure, throws an exception or returns Y_HARDWAREID_INVALID.

files→get_list() YFiles
files→list()

Returns a list of YFileRecord objects that describe files currently loaded in the filesystem.

```
js      function get_list( pattern)
nodejs  function get_list( pattern)
php     function get_list( $pattern)
cpp     vector<YFileRecord> get_list( string pattern)
m       -(NSMutableArray*) list : (NSString*) pattern
pas     function get_list( pattern: string): TYFileRecordArray
vb      function get_list( ) As List
cs      List<YFileRecord> get_list( string pattern)
java    ArrayList<YFileRecord> get_list( String pattern)
py      def get_list( pattern)
cmd     YFiles target get_list pattern
```

Parameters :

pattern an optional filter pattern, using star and question marks as wildcards. When an empty pattern is provided, all file records are returned.

Returns :

a list of `YFileRecord` objects, containing the file path and name, byte size and 32-bit CRC of the file content.

On failure, throws an exception or returns an empty list.

files→get_logicalName() YFiles
files→logicalName()

Returns the logical name of the filesystem.

```
js      function get_logicalName( )
nodejs  function get_logicalName( )
php     function get_logicalName( )
cpp     string get_logicalName( )
m       -(NSString*) logicalName
pas     function get_logicalName( ): string
vb      function get_logicalName( ) As String
cs      string get_logicalName( )
java    String get_logicalName( )
py      def get_logicalName( )
cmd     YFiles target get_logicalName
```

Returns :

a string corresponding to the logical name of the filesystem. On failure, throws an exception or returns `Y_LOGICALNAME_INVALID`.

files→get_module() YFiles
files→module()

Gets the YModule object for the device on which the function is located.

```
js      function get_module( )
nodejs  function get_module( )
php     function get_module( )
cpp     YModule * get_module( )
m       -(YModule*) module
pas     function get_module( ): TYModule
vb      function get_module( ) As YModule
cs      YModule get_module( )
java    YModule get_module( )
py      def 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)
nodejs  function get_module_async( callback, context)
```

If the function cannot be located on any module, the returned YModule object does not show as online. This asynchronous version exists only in Javascript. It uses a callback instead of a return value in order to avoid blocking Firefox javascript VM that does not implement context switching during blocking I/O calls. See the documentation section on asynchronous Javascript calls for more details.

Parameters :

callback callback function that is invoked when the result is known. The callback function receives three arguments: the caller-specific context object, the receiving function object and the requested YModule object

context caller-specific object that is passed as-is to the callback function

Returns :

nothing : the result is provided to the callback.

files→get_userData() YFiles
files→userData()

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

```
js      function get_userData( )
nodejs  function get_userData( )
php     function get_userData( )
cpp     void * get_userData( )
m       -(void*) userData
pas     function get_userData( ): Tobject
vb      function get_userData( ) As Object
cs      object get_userData( )
java    Object get_userData( )
py      def 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( )
nodejs  function isOnline( )
php     function isOnline( )
cpp     bool isOnline( )
m       -(BOOL) isOnline
pas     function isOnline( ): boolean
vb      function isOnline( ) As Boolean
cs      bool isOnline( )
java    boolean isOnline( )
py      def 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)
nodejs  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→load() YFiles

Preloads the filesystem cache with a specified validity duration.

```
js      function load( msValidity)
nodejs  function load( msValidity)
php     function load( $msValidity)
cpp     YRETCODE load( int msValidity)
m       -(YRETCODE) load : (int) msValidity
pas     function load( msValidity: integer): YRETCODE
vb      function load( ByVal msValidity As Integer) As YRETCODE
cs      YRETCODE load( int msValidity)
java    int load( long msValidity)
py      def 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 trafic 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→load_async() YFiles

Preloads the filesystem cache with a specified validity duration (asynchronous version).

```
js      function load_async( msValidity, callback, context)
nodejs  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 trafic 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→nextFiles() YFiles

Continues the enumeration of filesystems started using `yFirstFiles()`.

```
js      function nextFiles( )
nodejs  function nextFiles( )
php     function nextFiles( )
cpp     YFiles * nextFiles( )
m       -(YFiles*) nextFiles
pas     function nextFiles( ): TYFiles
vb      function nextFiles( ) As YFiles
cs      YFiles nextFiles( )
java    YFiles nextFiles( )
py      def nextFiles( )
```

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)
nodejs  function registerValueCallback( callback)
php     function registerValueCallback( $callback)
cpp     int registerValueCallback( YFilesValueCallback callback)
m       -(int) registerValueCallback : (YFilesValueCallback) callback
pas     function registerValueCallback( callback: TYFilesValueCallback): LongInt
vb      function registerValueCallback( ) As Integer
cs      int registerValueCallback( ValueCallback callback)
java    int registerValueCallback( UpdateCallback callback)
py      def 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)
nodejs  function remove( pathname)
php     function remove( $pathname)
cpp     int remove( string pathname)
m       -(int) remove : (NSString*) pathname
pas     function remove( pathname: string): LongInt
vb      function remove( ) As Integer
cs      int remove( string pathname)
java    int remove( String pathname)
py      def remove( pathname)
cmd     YFiles target remove pathname
```

Because of filesystem fragmentation, deleting a file may not always free up the whole space used by the file. However, rewriting a file with the same path name will always reuse any space not freed previously. If you need to ensure that no space is taken by previously deleted files, you can use `format_fs` to fully reinitialize the filesystem.

Parameters :

pathname path and name of the file to remove.

Returns :

`YAPI_SUCCESS` if the call succeeds.

On failure, throws an exception or returns a negative error code.

files→set_logicalName() YFiles
files→setLogicalName()

Changes the logical name of the filesystem.

```
js      function set_logicalName( newval)
nodejs  function set_logicalName( newval)
php     function set_logicalName( $newval)
cpp     int set_logicalName( const string& newval)
m       -(int) setLogicalName : (NSString*) newval
pas     function 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)
py      def set_logicalName( 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)
nodejs  function set_userData( data)
php     function set_userData( $data)
cpp     void set_userData( void* data)
m       -(void) setUserData : (void*) data
pas     procedure 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      def 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→upload() YFiles

Uploads a file to the filesystem, to the specified full path name.

```
js      function upload( pathname, content)
nodejs  function upload( pathname, content)
php     function upload( $pathname, $content)
cpp     int upload( string pathname, string content)
m       -(int) upload : (NSString*) pathname
                      : (NSData*) content
pas     function upload( pathname: string, content: TByteArray): LongInt
vb      procedure upload( )
cs      int upload( string pathname)
java    int upload( String pathname)
py      def upload( pathname, 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)
nodejs  function 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.

22. Troubleshooting

22.1. 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[1] archive, there are two rule examples which you can use as a basis.

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

[1] http://www.yoctopuce.com/FR/virtualhub.php

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"
```

22.2. ARM Platforms: HF and EL

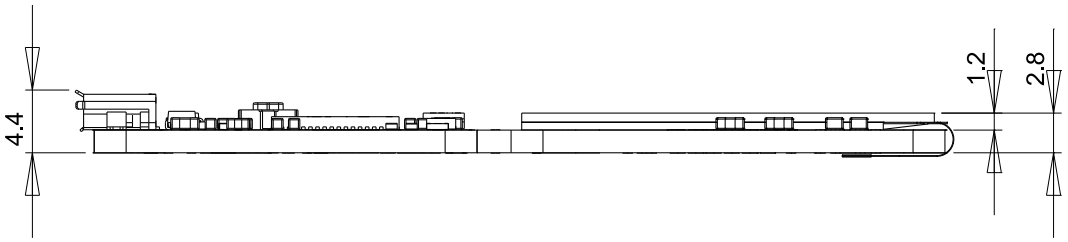
There are two main flavors of executable on ARM: HF (Hard Float) binaries, and EL (EABI Little Endian) binaries. These two families are not compatible at all. The compatibility of a given ARM platform with of one of these two families depends on the hardware and on the OS build. ArmHL and ArmEL compatibility problems are quite difficult to detect. Most of the time, the OS itself is unable to make a difference between an HF and an EL executable and will return meaningless messages when you try to use the wrong type of binary.

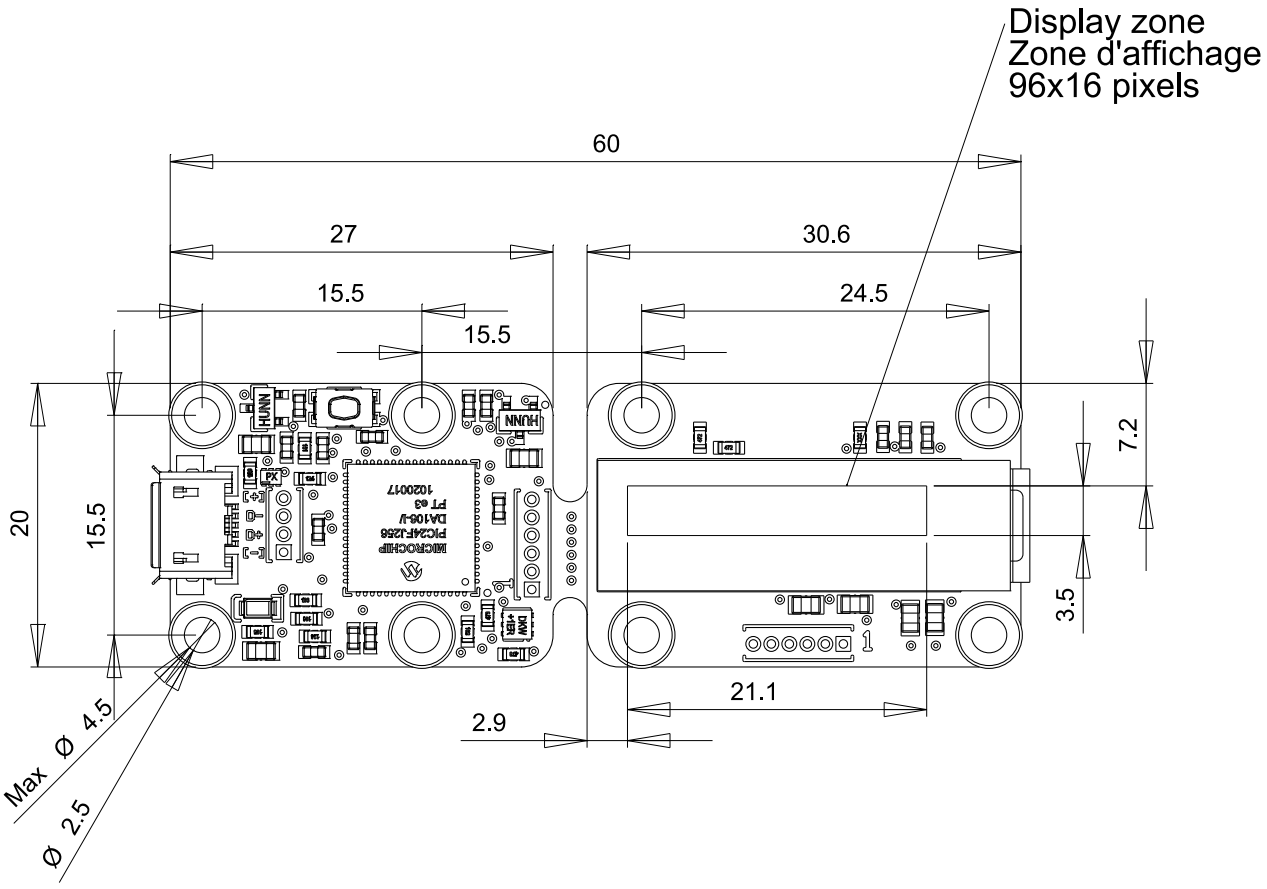
All pre-compiled Yoctopuce binaries are provided in both formats, as two separate ArmHF et ArmEL executables. If you do not know what family your ARM platform belongs to, just try one executable from each family.

23. Characteristics

You can find below a summary of the main technical characteristics of your Yocto-MiniDisplay module.

Resolution	96 x 16 px
Display area	21.1 x 3.5 mm
Width	20 mm
Length	60 mm
Weight	5 g
USB connector	micro-B
Supported Operating Systems	Windows, Linux (Intel + ARM), Mac OS X, Android
Drivers	no driver needed
API / SDK / Libraries (USB+TCP)	C++, Objective-C, C#, VB .NET, Delphi, Python, Java/Android
API / SDK / Libraries (TCP only)	Javascript, Node.js, PHP, Java
RoHS	yes
USB Vendor ID	0x24E0
USB Device ID	0x002F
Suggested enclosure	YoctoBox-Long-Thin-Black





All dimensions are in mm
Toutes les dimensions sont en mm

Yocto-MiniDisplay

A4

Scale
2:1
Echelle

Index

H

I

J

L

V

W

Y