



Yocto-3D-V2-C

User's guide

Table of contents

1. Introduction	1
1.1. <i>Safety Information</i>	2
1.2. <i>Environmental conditions</i>	3
2. Presentation	5
2.1. <i>Common elements</i>	5
2.2. <i>Specific elements</i>	6
3. The Yocto-3D-V2-C functions	7
3.1. <i>The 3D accelerometer</i>	7
3.2. <i>The 3D magnetometer</i>	7
3.3. <i>The 3D gyroscope</i>	8
3.4. <i>The reference frame</i>	8
3.5. <i>The 2D inclinometer (tilt sensor)</i>	9
3.6. <i>The magnetic compass</i>	10
3.7. <i>The estimated attitude quaternion</i>	10
3.8. <i>Optional accessories</i>	10
4. First steps	13
4.1. <i>Prerequisites</i>	13
4.2. <i>Testing USB connectivity</i>	14
4.3. <i>Localization</i>	15
4.4. <i>Test of the module</i>	15
4.5. <i>Configuration</i>	16
5. Assembly and connections	19
5.1. <i>Fixing</i>	19
5.2. <i>Moving the sensor away</i>	20
5.3. <i>USB power distribution</i>	21
5.4. <i>Electromagnetic compatibility (EMI)</i>	22
6. Programming, general concepts	23
6.1. <i>Programming paradigm</i>	23
6.2. <i>The Yocto-3D-V2-C module</i>	25

6.3. Module	28
6.4. Accelerometer	29
6.5. Magnetometer	30
6.6. Gyro	31
6.7. RefFrame	33
6.8. Tilt	33
6.9. Compass	35
6.10. DataLogger	36
6.11. What interface: Native, DLL or Service ?	37
6.12. Accessing modules through a hub	39
6.13. Programming, where to start?	40
7. Using the Yocto-3D-V2-C in command line	41
7.1. Installing	41
7.2. Use: general description	41
7.3. Control of the Tilt function	42
7.4. Control of the module part	42
7.5. Limitations	43
8. Using the Yocto-3D-V2-C with Python	45
8.1. Source files	45
8.2. Dynamic library	45
8.3. Control of the Tilt function	45
8.4. Control of the module part	48
8.5. Error handling	49
9. Using Yocto-3D-V2-C with C++	51
9.1. Control of the Tilt function	51
9.2. Control of the module part	54
9.3. Error handling	56
9.4. Integration variants for the C++ Yoctopuce library	57
10. Using Yocto-3D-V2-C with C#	59
10.1. Installation	59
10.2. Using the Yoctopuce API in a Visual C# project	59
10.3. Control of the Tilt function	60
10.4. Control of the module part	62
10.5. Error handling	65
11. Using the Yocto-3D-V2-C with LabVIEW	67
11.1. Architecture	67
11.2. Compatibility	68
11.3. Installation	68
11.4. Presentation of Yoctopuce VIs	73
11.5. Functioning and use of VIs	76
11.6. Using	78
11.7. Managing the data logger	80
11.8. Function list	81
11.9. A word on performances	82
11.10. A full example of a LabVIEW program	82
11.11. Differences from other Yoctopuce APIs	83

12. Using the Yocto-3D-V2-C with Java	85
12.1. <i>Getting ready</i>	85
12.2. <i>Control of the Tilt function</i>	85
12.3. <i>Control of the module part</i>	87
12.4. <i>Error handling</i>	90
13. Using the Yocto-3D-V2-C with Android	91
13.1. <i>Native access and VirtualHub</i>	91
13.2. <i>Getting ready</i>	91
13.3. <i>Compatibility</i>	91
13.4. <i>Activating the USB port under Android</i>	92
13.5. <i>Control of the Tilt function</i>	93
13.6. <i>Control of the module part</i>	96
13.7. <i>Error handling</i>	101
14. Using Yocto-3D-V2-C with TypeScript	103
14.1. <i>Using the Yoctopuce library for TypeScript</i>	104
14.2. <i>Refresher on asynchronous I/O in JavaScript</i>	104
14.3. <i>Control of the Tilt function</i>	105
14.4. <i>Control of the module part</i>	108
14.5. <i>Error handling</i>	110
15. Using Yocto-3D-V2-C with JavaScript / EcmaScript	113
15.1. <i>Blocking I/O versus Asynchronous I/O in JavaScript</i>	113
15.2. <i>Using Yoctopuce library for JavaScript / EcmaScript 2017</i>	114
15.3. <i>Control of the Tilt function</i>	116
15.4. <i>Control of the module part</i>	120
15.5. <i>Error handling</i>	122
16. Using Yocto-3D-V2-C with PHP	125
16.1. <i>Getting ready</i>	125
16.2. <i>Control of the Tilt function</i>	126
16.3. <i>Control of the module part</i>	128
16.4. <i>HTTP callback API and NAT filters</i>	130
16.5. <i>Error handling</i>	134
17. Using Yocto-3D-V2-C with Visual Basic .NET	135
17.1. <i>Installation</i>	135
17.2. <i>Using the Yoctopuce API in a Visual Basic project</i>	135
17.3. <i>Control of the Tilt function</i>	136
17.4. <i>Control of the module part</i>	138
17.5. <i>Error handling</i>	140
18. Using Yocto-3D-V2-C with Delphi or Lazarus	143
18.1. <i>Preparation</i>	143
18.2. <i>About examples</i>	144
18.3. <i>Control of the Tilt function</i>	144
18.4. <i>Control of the module part</i>	147
18.5. <i>Error handling</i>	149
19. Using the Yocto-3D-V2-C with Universal Windows Platform	151

19.1. Blocking and asynchronous functions	151
19.2. Installation	152
19.3. Using the Yoctopuce API in a Visual Studio project	152
19.4. Control of the Tilt function	153
19.5. A real example	154
19.6. Control of the module part	155
19.7. Error handling	157
20. Using Yocto-3D-V2-C with Objective-C	159
20.1. Control of the Tilt function	159
20.2. Control of the module part	162
20.3. Error handling	164
21. Using with unsupported languages	165
21.1. Command line	165
21.2. .NET Assembly	165
21.3. VirtualHub and HTTP GET	167
21.4. Using dynamic libraries	169
21.5. Porting the high level library	172
22. Firmware Update	173
22.1. VirtualHub or the YoctoHub	173
22.2. The command line library	173
22.3. The Android application Yocto-Firmware	173
22.4. Updating the firmware with the programming library	174
22.5. The "update" mode	176
23. Advanced programming	177
23.1. Event programming	177
23.2. The data logger	180
24. High-level API Reference	183
24.1. Class YAPI	184
24.2. Class YModule	187
24.3. Class YAccelerometer	192
24.4. Class YMagnetometer	197
24.5. Class YRefFrame	202
24.6. Class YTilt	207
24.7. Class YCompass	212
24.8. Class YGyro	217
24.9. Class YDataLogger	223
24.10. Class YDataSet	227
24.11. Class YMeasure	230
25. Troubleshooting	231
25.1. Where to start?	231
25.2. Programming examples don't seem to work	231
25.3. Linux and USB	231
25.4. ARM Platforms: HF and EL	232
25.5. Powered module but invisible for the OS	232
25.6. Another process named xxx is already using yAPI	232
25.7. Disconnections, erratic behavior	232

25.8. After a failed firmware update, the device stopped working	233
25.9. The web interface shows errors after a firmware update	233
25.10. Registering VirtualHub disconnects another instance	233
25.11. Dropped commands	233
25.12. Damaged device	233
26. Characteristics	235
<i>Blueprint</i>	238

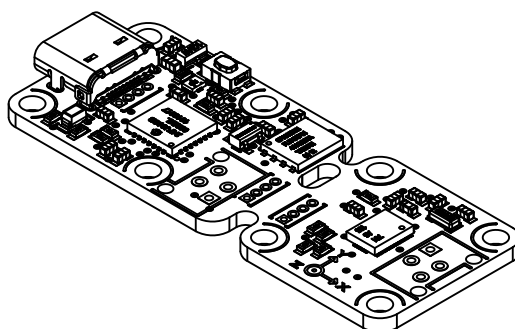
1. Introduction

The Yocto-3D-V2-C is a 51x20mm electronic module which contains an accelerometer, a gyroscope, and a magnetometer. The three sensors are three-dimensional. The Yocto-3D-V2-C uses these three sensors to measure the inclination with regards to the horizontal plane (tilt sensor), the magnetic heading (tilt-compensated compass), the acceleration, and the magnetic field to which the module is subjected as well as the angular velocity on each axis.

Moreover, the module is able to instantly estimate its orientation by combining the integration of gyroscopic measures with the static orientation measurements. The device attitude is provided directly as a quaternion, and as aeronautical angles (using Tait-Bryan's angles: roll, pitch, and heading). No extra math is required in the application to get this information.

Contrarily to the first version of the Yocto-3D, this version 2 is capable of performing constant auto-calibration, in particular to enable the use of the compass even in presence of hard-iron distortions, caused by magnetic sources that move together with the Yocto-3D-V2-C. However, soft-iron distortions, caused by ferromagnetic materials near to the sensor which alter the magnetic field, are not compensated for.

A flash memory is provided in the module, enabling it to autonomously store the measures for later retrieval, and to save the orientation sensor calibration so that it is functional immediately at power-on.



The Yocto-3D-V2-C module

The Yocto-3D-V2-C is not in itself a complete product. It is a component intended to be integrated into a solution used in laboratory equipments, or in industrial process-control equipments, or for similar applications in domestic and commercial environments. In order to use it, you must at least install it in a protective enclosure and connect it to a host computer.

Yoctopuce thanks you for buying this Yocto-3D-V2-C and sincerely hopes that you will be satisfied with it. The Yoctopuce engineers have put a large amount of effort to ensure that your Yocto-3D-V2-

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

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

1.1. Safety Information

The Yocto-3D-V2-C is designed to meet the requirements of IEC 61010-1:2010 safety standard. It does not create any serious hazards to the operator and surrounding area, even in single fault condition, as long as it is integrated and used according to the instructions contained in this documentation, and in this section in particular.

Protective enclosure

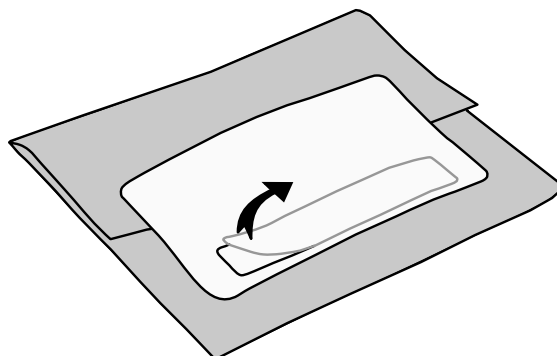
The Yocto-3D-V2-C should not be used without a protective enclosure, because of the accessible bare electronic components. For optimal safety, it should be put into a non-metallic, non-inflammable enclosure, resistant to a mechanical stress level of 5 J. For instance, use a polycarbonate (e.g. LEXAN) enclosure rated IK08 with a IEC 60695-11-10 flammability rating of V-1 or better. Using a lower quality enclosure may require specific warnings for the operator and/or compromise conformity with the safety standard.

Maintenance

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

Identification

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



Identification label is integrated in the package label.

Application

The safety standard applied is intended to cover laboratory equipment, industrial process-control equipment and similar applications in residential or commercial environment. If you intend to use the Yocto-3D-V2-C for another kind of application, you should check the safety regulations according to the standard applicable to your application.

In particular, the Yocto-3D-V2-C is *not* certified for use in medical environments or for life-support applications.

Environment

The Yocto-3D-V2-C is *not* certified for use in hazardous locations, explosive environments, or life-threatening applications. Environmental ratings are provided below.

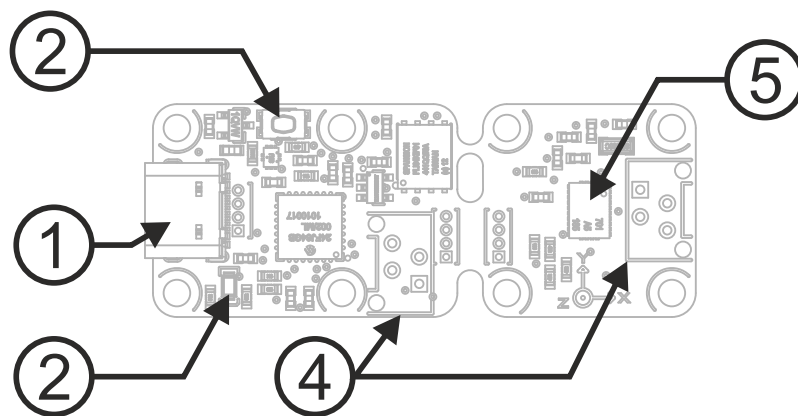
1.2. Environmental conditions

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

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

The extended operating temperature range for the Yocto-3D-V2-C is -30...85°C. This temperature range has been determined based on components manufacturer recommendations, and on controlled environment tests performed during a limited duration (1h). If you plan to use the Yocto-3D-V2-C in harsh environments for a long period of time, we strongly advise you to run extensive tests before going to production.

2. Presentation



1: USB-C socket 4: Picoflex header locations
2: Yocto-button 5: Sensor
3: Yocto-led

2.1. Common elements

All Yocto-modules share a number of common functionalities.

USB connector

The Yocto-3D-V2-C features an USB Type-C socket and uses USB 1.1 protocol which makes it compatible with any USB host port. Alternatively USB wires or a 1.27mm connector can be soldered on the footprint right behind the USB socket.

If, when you plug your Yocto-3D-V2-C into a USB-C cable, your module lights up but is not detected by your computer, check that the USB-C plug is fully inserted and that you're using a regular USB cable, not a charging cable with no data lines.

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

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

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

Yocto-button

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

Yocto-led

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

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

The Yocto-led has a third functionality, which is less pleasant: when the internal software which controls the module encounters a fatal error, the Yocto-led starts emitting an SOS in morse ¹. If this happens, unplug and re-plug the module. If it happens again, check that the module contains the latest version of the firmware, and, if it is the case, contact Yoctopuce support².

Current sensor

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

Serial number

Each Yocto-module has a unique serial number assigned to it at the factory. For Yocto-3D-V2-C modules, this number starts with Y3DMK02C. 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 sensors

The Yocto-3D-V2-C uses a BNO055 sensor manufactured by Bosch. It is a 3D accelerometer coupled with a 3D magnetometer and 3D gyro. This sensor handles internally the fusion of all sensors in real time, and performs continuous auto-calibration. Be aware that it is a relatively fragile component: an acceleration above 2000g for more than 1 [ms], or an acceleration of 10000g for more than 0.2 [ms] would damage the sensor. For instance, a free fall onto a hard surface from a height above 1.80m could damage the sensor.

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

² support@yoctopuce.com

3. The Yocto-3D-V2-C functions

The Yocto-3D-V2-C has many distinct functions, providing various interfaces to its inertial sensors as may be required by the target application.

3.1. The 3D accelerometer

This function shows the acceleration to which the module is submitted, in multiples of the gravitational acceleration (g). The main value is the norm of the acceleration vector, but you can obtain X, Y, and Z components individually. The maximal measured value is of about 16g. Be aware that the Yocto-3D-V2-C accelerometer may detect a calibration loss in case an acceleration over 16g is detected. This would result in an immediate reduction of capabilities on all Yocto-3D-V2-C functions which depend directly or indirectly on the orientation estimate.

On the Yocto-3D-V2-C, it is possible to configure the acceleration report frequency between 1Hz and 100Hz. Internally, the measure is sampled at 100Hz, but when a lower frequency is selected, measures are averaged over the selected period.

To obtain a very precise value taking into account the gravitational force at the location where the module is used, you can perform a semi-automatic calibration of the accelerometer.

3.2. The 3D magnetometer

This function shows the magnetic field intensity to which the module is subjected, in Gauss. The main value is the field vector, but you can obtain the X, Y, and Z values individually. To be as precise as possible to detect the earth magnetic field, the magnetometer is limited to +/- 2 Gauss.

Beware, this function is easily perturbed in the proximity of ferromagnetic objects (such as steel screws) which modify the field lines.

Contrarily to the first version of the Yocto-3D, this version 2 is capable of performing constant auto-calibration, in particular to enable the use of the compass even in presence of hard-iron distortions, caused by magnetic sources that move together with the Yocto-3D-V2-C. However, soft-iron distortions, caused by ferromagnetic materials near to the sensor which alter the magnetic field, are not compensated for. If you intend to use the magnetometer function, it is therefore important to use only non-ferrous metals near to the sensor. For instance, aluminium, copper, zinc, brass and silver are good. Make sure to avoid iron, nickel, steel and cobalt.

On the Yocto-3D-V2-C, it is possible to configure the magnetometer report frequency between 1Hz and 20Hz. Internally, the measure is sampled at 20Hz, but when a lower frequency is selected, measures are averaged over the selected period.

3.3. The 3D gyroscope

This function shows the angular velocity of the device, in degrees per second. The main value is the norm of the angular velocity vector, but you can also obtain the angular velocity around individual axis X, Y, and Z. The maximal measured value is about 2000 degrees per second.

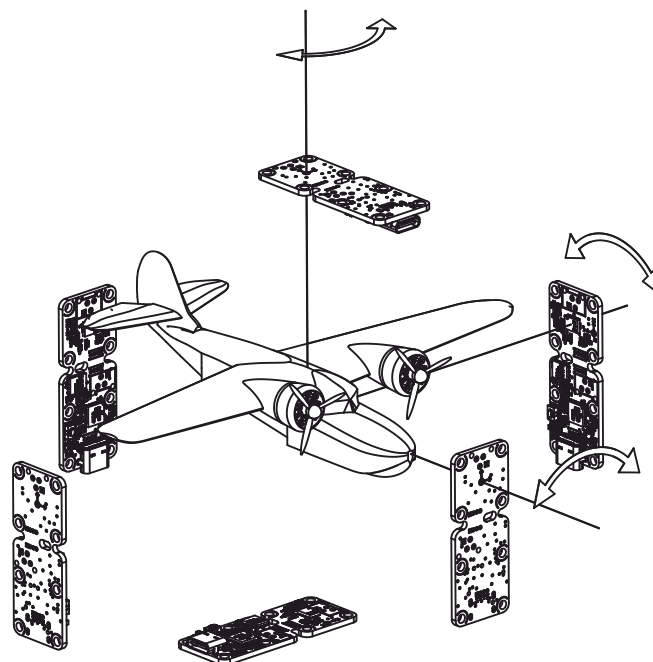
In order to counter the natural drift of the gyroscope, the Yocto-3D-V2-C automatically zeroes the measured value when the device detects that it is not moving (acceleration vector staying still and around 1g for several tenths of seconds at least).

On the Yocto-3D-V2-C, it is possible to configure the gyroscope report frequency between 1Hz and 100Hz. Internally, the measure is sampled at 100Hz, but when a lower frequency is selected, measures are averaged over the selected period.

3.4. The reference frame

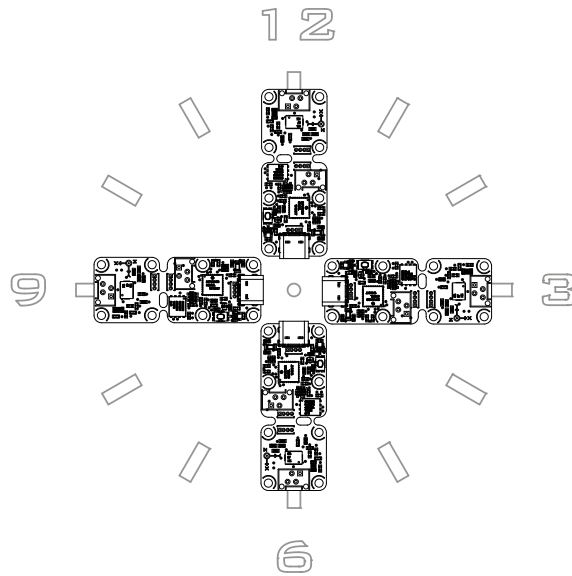
You can freely choose the reference frame in which the Yocto-3D-V2-C is going to work, that is the orientation in which it is assembled with regards to the natural movement direction. Indeed, the module orientation intervenes directly in the tilt measure computations (which are measured with regards to the horizontal plane), as well as for angle measures. In all, there are 24 orthogonal references you can select from. They can be simplified into 6 groups of positions (corresponding to the 6 sides of a cube), each including 4 distinct references for the 4 possible 90° rotations around the module Z axis.

Let us imagine that the Yocto-3D-V2-C is assembled into a plane. It can be fixed on the left wall (LEFT), on the right wall (RIGHT), on the floor (BOTTOM), on the ceiling (TOP), at the front (FRONT), or at the rear (REAR).



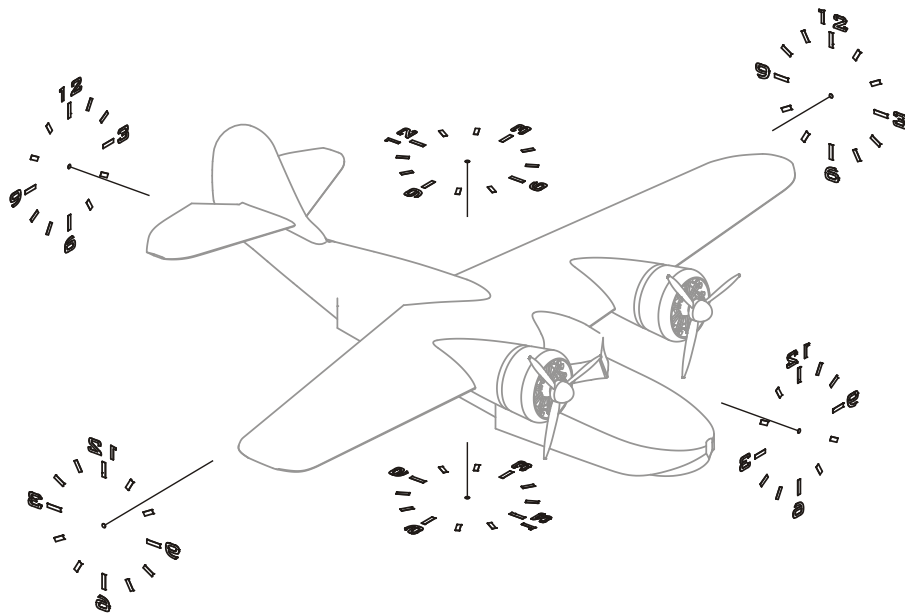
The module 6 assembly positions

For each assembly position, there are 4 orthogonal orientations, which are defined by analogy with a clock: 12:00, 3:00, 6:00, and 9:00.



The 4 assembly orientations

The orientation of each of the 6 clocks is defined in a natural way with regards to what the pilot of the plane would see.



Orientation of the six clocks

Make sure to configure the Yocto-3D-V2-C orientation after it is assembled, as all functions listed below in this sections depend on this settings. Do not forget to save this configuration in the module flash memory. Otherwise, the Yocto-3D-V2-C will take the default configuration (BOTTOM, TWELVE) the next time it starts.

Whenever you change the configuration of the Yocto-3D-V2-C orientation, make sure to rerun an explicit calibration procedure to ensure that the calibration values saved in the flash memory correspond to the selected orientation.

3.5. The 2D inclinometer (tilt sensor)

The Yocto-3D-V2-C can measure the tilt with regards to the horizontal plane, with the help of a gravity measure. You can configure the choice of the reference horizontal plane in the `refFram` function described above. By default, it is the module X/Y plane.

The `tilt1` function corresponds to roll, while the `tilt2` function corresponds to pitch. Roll is a positive angle when the left side of the module rises. And pitch is a positive angle when the front of the module rises.

On the Yocto-3D-V2-C, it is possible to configure the inclinometer report frequency between 1Hz and 100Hz. Internally, the measure is sampled at 100Hz, but when a lower frequency is selected, measures are averaged over the selected period.

3.6. The magnetic compass

This function provides the module bearing, such as determined thanks to the earth magnetic field (as with a compass). The bearing is an angle in degrees, from 0° to 360°, clockwise. The magnetic field measure is compensated to take into account the module tilt. The module does not need, therefore, to be horizontal for the value to be correct. In case you observe a bearing change when the device is tilted, it probably means that you are experiencing soft iron distortions due to some nearby ferrous metal.

By default, bearing is computed on the module X/Y plane, but this plan can be changed by modifying the orientation defined with the `refFrame` function. Likewise, by default, 0° corresponds to the magnetic North, but it is possible to configure a different reference angle, for instance to compensate for the local magnetic declination and to point to the geographic North.

Beware, this function is easily perturbed in the proximity of ferromagnetic objects (such as steel screws) which modify the field lines.

Contrarily to the first version of the Yocto-3D, this version 2 is capable of performing constant auto-calibration, in particular to enable the use of the compass even in presence of hard-iron distortions, caused by magnetic sources that move together with the Yocto-3D-V2-C. However, soft-iron distortions, caused by ferromagnetic materials near to the sensor which alter the magnetic field, are not compensated for. If you intend to use the magnetometer function, it is therefore important to use only non-ferrous metals near to the sensor. For instance, aluminium, copper, zinc, brass and silver are good. Make sure to avoid iron, nickel, steel and cobalt.

On the Yocto-3D-V2-C, it is possible to configure the compass report frequency between 1Hz and 20Hz. Internally, the measure is sampled at 20Hz, but when a lower frequency is selected, measures are averaged over the selected period.

3.7. The estimated attitude quaternion

The four functions `qt1` to `qt4` correspond to the four components of the hypercomplex quaternion (also known as w, x, y and z) describing the estimated attitude of the device, based on a 95Hz gyroscopic integration. To counter the natural drift of the integration process, the Yocto-3D-V2-C combines it with tilt and compass measurements using a complimentary filter.

The `YGyro` class of the programming library provides a simple API to access the quaternion, including its conversion into Tait-Bryan's roll, pitch, and heading angles). In the Yocto-3D-V2-C, angles returned by the inclinometers and the magnetic compass correspond directly to the values that can be derived from the quaternion.

3.8. Optional accessories

The accessories below are not necessary to use the Yocto-3D-V2-C module but might be useful depending on your project. These are mostly common products that you can buy from your favorite DIY store. To save you the tedious job of looking for them, most of them are also available on the Yoctopuce shop.

Screws and spacers

In order to mount the Yocto-3D-V2-C module, you can put small screws in the 2.5mm assembly holes, with a screw head no larger than 4.5mm. The best way is to use threaded spacers, which you

can then mount wherever you want. You can find more details on this topic in the chapter about assembly and connections.

Micro-USB hub

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

YoctoHub-Ethernet, YoctoHub-Wireless and YoctoHub-GSM

You can add network connectivity to your Yocto-3D-V2-C, thanks to the YoctoHub-Ethernet, the YoctoHub-Wireless and the YoctoHub-GSM which provides respectively Ethernet, WiFi and GSM connectivity. All of them can drive up to three devices and behave exactly like a regular computer running the VirtualHub application¹.

1.27mm (or 1.25mm) connectors

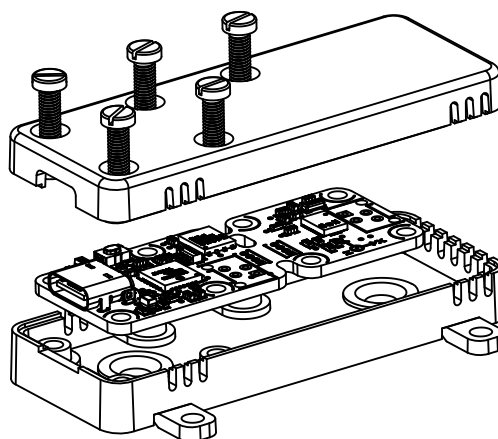
In case you wish to connect your Yocto-3D-V2-C to a Micro-hub USB or a YoctoHub without using a bulky USB connector, you can use the four 1.27mm pads just behind the USB connector. There are two options.

You can mount the Yocto-3D-V2-C directly on the hub using screw and spacers, and connect it using 1.27mm board-to-board connectors. To prevent shortcuts, it is best to solder the female connector on the hub and the male connector on the Yocto-3D-V2-C.

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

Enclosure

Your Yocto-3D-V2-C has been designed to be installed as is in your project. Nevertheless, Yoctopuce sells enclosures specifically designed for Yoctopuce devices. These enclosures have removable mounting brackets and magnets allowing them to stick on ferromagnetic surfaces. More details are available on the Yoctopuce web site². The suggested enclosure model for your Yocto-3D-V2-C is the YoctoBox-3D-Black.



You can install your Yocto-3D-V2-C in an optional enclosure

¹ <http://www.yoctopuce.com/EN/virtualhub.php>

² <http://www.yoctopuce.com/EN/products/category/enclosures>

Picoflex connectors and flexible ribbon cable

If you intend to move the sensor component away from the rest of the Yocto-3D-V2-C module using a pluggable cable, you will need 4-wire ribbon cable of 1.27mm pitch, and Picoflex connectors.³ You can find more details on this topic in the chapter about assembly and connections.

Be aware that picoflex connectors are ferromagnetic and may alter the compass function. Only use it on the sensor side if you only need accelerometer and gyroscope functions. If you need the compass to work properly, you should solder the ribbon cable directly on the PCB on the sensor side.

³ Header Molex ref 90325-3004 or 90325-0004, available from most electronic components suppliers (www.mouser.com, www.digikey.com, www.farnell.com, www.distrelec.ch...). To be used with connectors ref 90327-3304 or 90327-0304.

4. First steps

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

4.1. Prerequisites

In order to use your Yocto-3D-V2-C module, you should have the following items at hand.

A computer

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

Yoctopuce provides software libraries to drive its modules for the following operating systems: **Windows, Linux, macOS, and Android**. Yoctopuce modules do not require the installation of specific drivers, as they use the HID driver¹ standardly supplied in all operating systems.

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

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

The architectures supported by Yoctopuce software libraries are as follows:

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

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

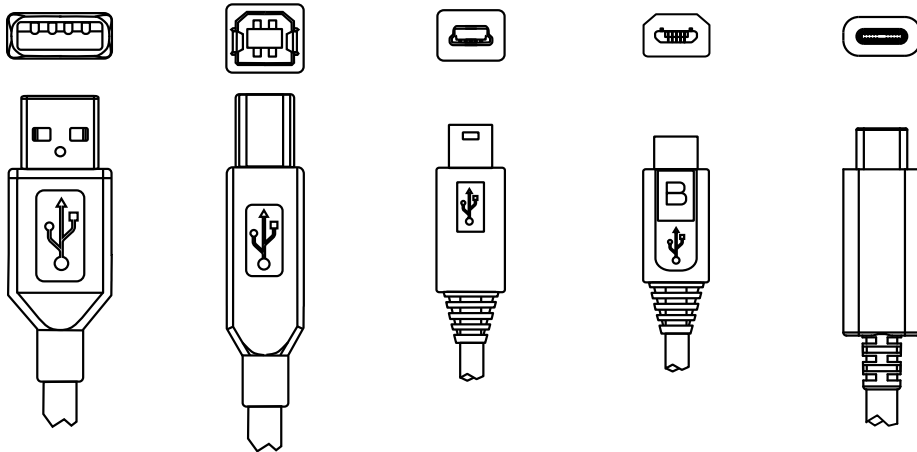
¹ The HID driver is manages peripheral devices such as mouse, keyboard, and so on.

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

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

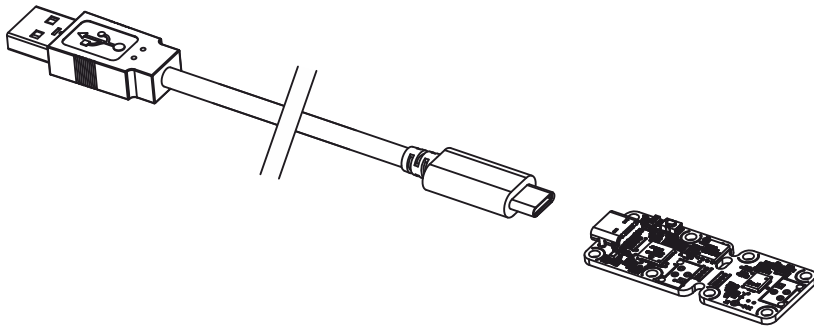
An USB cable, type A - USB-C

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



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

To connect your Yocto-3D-V2-C module to a computer, you need an USB cable of type A-USB-C or type C-USB-C. The price of this cable may vary a lot depending on the source, look for it under the name *USB A to USB-C Data cable*. Make sure not to buy a simple USB charging cable without data connectivity. The correct type of cable is available on the Yoctopuce shop.



You must plug in your Yocto-3D-V2-C module with a USB 2.0 cable of type A-USB-C

If you insert a USB hub between the computer and the Yocto-3D-V2-C module, make sure to take into account the USB current limits. If you do not, be prepared to face unstable behaviors and unpredictable failures. You can find more details on this topic in the chapter about assembly and connections.

4.2. Testing USB connectivity

At this point, your Yocto-3D-V2-C should be connected to your computer, which should have recognized it. It is time to make it work.

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

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

Serial	Logical Name	Description	Action
VIRTHUB2-3880db7f12		VirtualHub-V2	configure view log file
└─Y3DMK02C-2DE50D		Yocto-3D-V2-C	configure view log file beacon

Module list as displayed in your web browser

4.3. Localization

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

4.4. Test of the module

The first item to check is that your module is working well: click on the serial number corresponding to your module. This displays a window summarizing the properties of your Yocto-3D-V2-C.

Y3DMK02C-2DE50D



Y3DMK02C-2DE50D

orientation sensor

is a 20x51mm

Module

Serial #

Product name:

Logical name:

Firmware:

Consumption:

Beacon:

Luminosity:

Y3DMK02C-2DE50D

Yocto-3D-V2-C

68601

37 mA

Inactive

50%

turn on

Sensors

Tilt Roll:

Tilt Pitch:

Compass:

Accelerometer:

Accelerometer X:

Accelerometer Y:

Accelerometer Z:

Magnetometer:

Magnetometer X:

Magnetometer Y:

Magnetometer Z:

Gyro:

Gyro X:

Gyro Y:

Gyro Z:

-35.1 °

-5.1 °

92.6 °

1 g

-0.087 g

-0.58 g

0.81 g

13.851 gauss

-13.037 gauss

-4.49 gauss

-1.308 gauss

0.1 °/s

0 °/s

0 °/s

0.1 °/s

View 3D

View 3D

View 3D

View 3D

Misc

Open API browser

Get user manual from yoctopuce.com

Close

Properties of the Yocto-3D-V2-C module

This window allows you to play with your module to check how it is working. Values measured by the Yocto-3D-V2-C are indeed displayed in real time, both as numerical values and as a graph.

³ www.yoctopuce.com/EN/virtualhub.php

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

4.5. Configuration

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

Yocto-3D-V2-C module configuration

Firmware

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

To update a firmware, simply click on the **upgrade** button on the configuration window and follow the instructions. If the update fails for one reason or another, unplug and re-plug the module and start the update process again. This solves the issue in most cases. If the module was unplugged while it was being reprogrammed, it does probably not work anymore and is not listed in the interface. However, it is always possible to reprogram the module correctly by using VirtualHub⁵ in command line⁶.

Logical name of the module

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

Luminosity

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

⁵ www.yoctopuce.com/EN/virtualhub.php

⁶ More information available in the VirtualHub documentation

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 Yocto-3D-V2-C module has 9 functions. Simply click on the corresponding "rename" buttons to assign them new logical names.

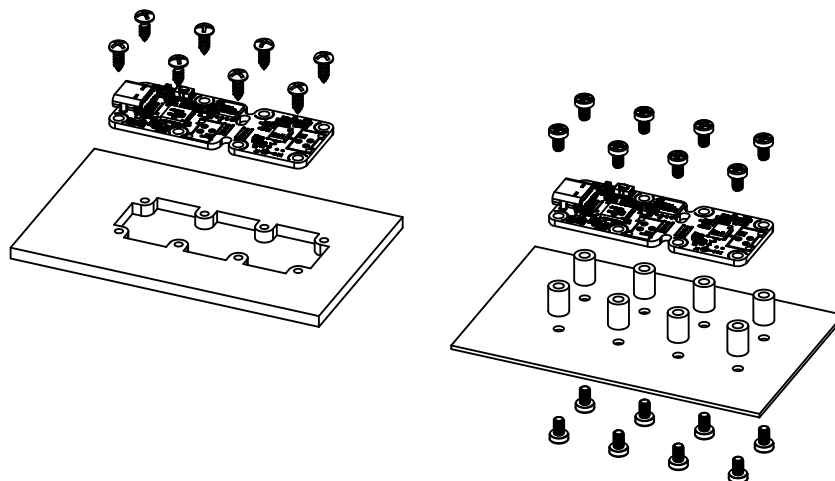
You can also configure the reference frame used by the module for the tilt sensors, the compass, and the gyroscopic estimate of the module orientation.

5. Assembly and connections

This chapter provides important information regarding the use of the Yocto-3D-V2-C module in real-world situations. Make sure to read it carefully before going too far into your project if you want to avoid pitfalls.

5.1. Fixing

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



Examples of assembly on supports

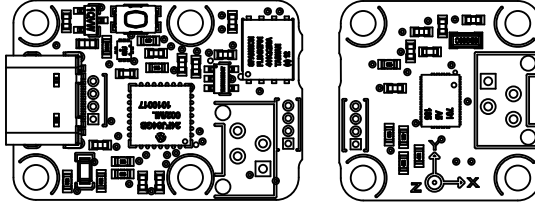
The Yocto-3D-V2-C module contains 2.5mm assembly holes. You can use these holes for screws. The screw head diameter must not be larger than 4.5mm or they will damage the module circuits. Make sure that the lower surface of the module is not in contact with the support. We recommend using spacers, but other methods are possible. Nothing prevents you from fixing the module with a glue gun; it will not be good-looking, but it will hold.

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

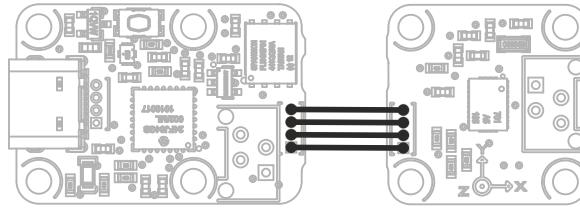
The use of mounting elements made of ferrous material, such as steel screws, is likely to prevent the magnetometer and the compass from working correctly.

5.2. Moving the sensor away

The Yocto-3D-V2-C module is designed so that you can split it into two parts, allowing you to move away the sensor from the command sub-module. You can split the module by simply breaking the circuit. However, you can obtain better results if you use a good pincer, or cutting pliers. When you have split the sub-modules, you can sandpaper the protruding parts without risk.

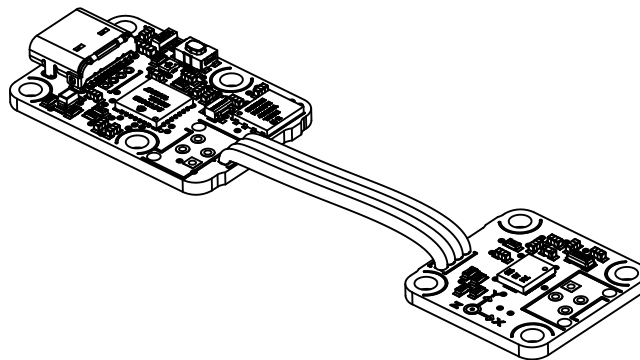


The Yocto-3D-V2-C module is designed so that you can split it into two parts.



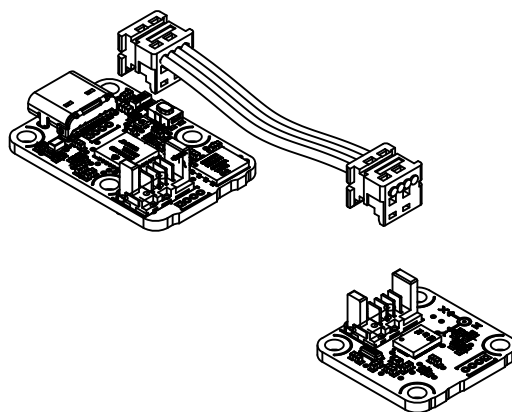
Wiring under the sub-modules once separated.

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.



Moving the sensor away with a ribbon cable.

You can also use ribbon cable with Picoflex connectors. You will obtain a slightly larger system, but the Picoflex headers are much easier to solder than ribbon cable. More over, the result can be disassembled. However, if you intend to use the magnetic compass function, make sure not to put a Picoflex connector on the sensor side. Picoflex connectors include ferrous metal which can alter the shape of magnetic field. It is therefore better to leave the Picoflex connector on the device side only, and solder the copper cable directly on the sensor side.



Moving the sensor away with Picoflex connectors.

Make sure to keep the sensor part away from ferromagnetic elements: a simple steel screw can considerably alter the magnetometer values, and therefore the compass.

Warning: divisible Yoctopuce modules very often have very similar connection systems. Nevertheless, sub-modules from different models are not all compatible. If you connect your Yocto-3D-V2-C sub-module to another type of module, such as a Yocto-Temperature for instance, it will not work, and you run a high risk of damaging your equipment.

5.3. 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 support.

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 protected enough current, they may work erratically and create unpredictable bugs. If you want to prevent any risk, do not cascade hubs without external power supply, and do not connect peripherals requiring more than 100mA behind a bus-powered hub.

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

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

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

5.4. Electromagnetic compatibility (EMI)

Connection methods to integrate the Yocto-3D-V2-C obviously have an impact on the system overall electromagnetic emissions, and therefore also impact the conformity with international standards.

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

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

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

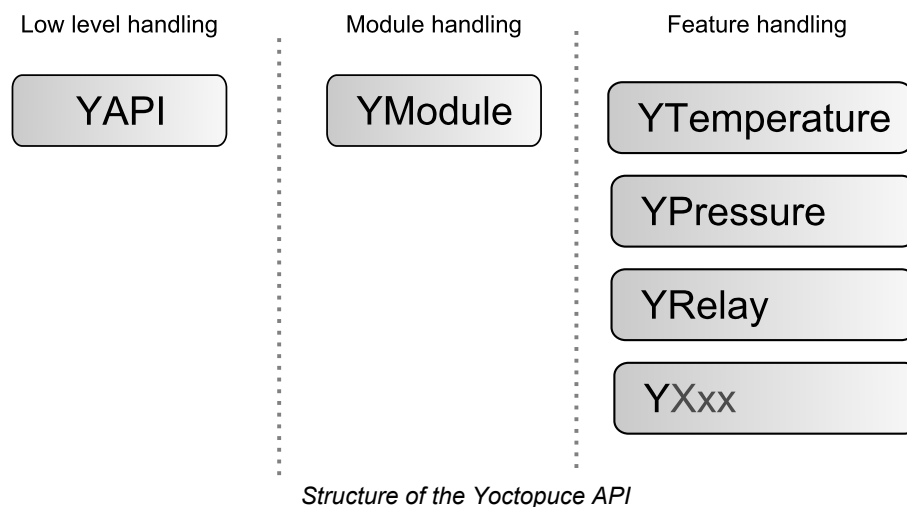
6. Programming, general concepts

The Yoctopuce API was designed to be at the same time simple to use and sufficiently generic for the concepts used to be valid for all the modules in the Yoctopuce range, and this in all the available programming languages. Therefore, when you have understood how to drive your Yocto-3D-V2-C with your favorite programming language, learning to use another module, even with a different language, will most likely take you only a minimum of time.

6.1. Programming paradigm

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

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



The YSensor class

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

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

Programmation

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

From the programming stand point, your Yocto-3D-V2-C is viewed as a module hosting a given number of functions. In the API, these functions are objects which can be found independently, in several ways.

Access to the functions of a module

Access by logical name

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

Access by enumeration

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

Access by hardware name

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

Difference between *Find* and *First*

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

Function handling

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

Access to the modules

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

Functions/Module interaction

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

6.2. The Yocto-3D-V2-C module

The Yocto-3D-V2-C device provides multiple measures. Raw sensor values can be obtained through the accelerometer, magnetometer and gyro interfaces. Computed inclination relative to the horizontal plane can be obtained through the tilt1 and tilt2 interfaces, as well as heading relative to the horizontal magnetic field. The device also provides an estimate of its 3D orientation based on the mathematical fusion of all sensors.

module : Module

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

accelerometer : Accelerometer

attribute	type	modifiable ?
logicalName	String	modifiable
advertisedValue	String	modifiable
unit	String	read-only
currentValue	Fixed-point number	read-only
lowestValue	Fixed-point number	modifiable
highestValue	Fixed-point number	modifiable
currentRawValue	Fixed-point number	read-only
logFrequency	Frequency	modifiable
reportFrequency	Frequency	modifiable
advMode	Enumerated	modifiable
calibrationParam	Calibration parameters	modifiable
resolution	Fixed-point number	modifiable
sensorState	Integer	read-only
bandwidth	Integer	modifiable
xValue	Fixed-point number	read-only
yValue	Fixed-point number	read-only
zValue	Fixed-point number	read-only
gravityCancellation	On/Off	modifiable

magnetometer : Magnetometer

attribute	type	modifiable ?
logicalName	String	modifiable

advertisedValue	String	modifiable
unit	String	read-only
currentValue	Fixed-point number	read-only
lowestValue	Fixed-point number	modifiable
highestValue	Fixed-point number	modifiable
currentRawValue	Fixed-point number	read-only
logFrequency	Frequency	modifiable
reportFrequency	Frequency	modifiable
advMode	Enumerated	modifiable
calibrationParam	Calibration parameters	modifiable
resolution	Fixed-point number	modifiable
sensorState	Integer	read-only
bandwidth	Integer	modifiable
xValue	Fixed-point number	read-only
yValue	Fixed-point number	read-only
zValue	Fixed-point number	read-only

gyro : Gyro

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

refFrame : RefFrame

attribute	type	modifiable ?
logicalName	String	modifiable
advertisedValue	String	modifiable
mountPos	Integer	modifiable
bearing	Fixed-point number	modifiable
calibrationParam	Calibration parameters	modifiable
fusionMode	Fusion mode	modifiable

tilt1 : Tilt**tilt2 : Tilt**

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

logFrequency	Frequency	modifiable
reportFrequency	Frequency	modifiable
advMode	Enumerated	modifiable
calibrationParam	Calibration parameters	modifiable
resolution	Fixed-point number	modifiable
sensorState	Integer	read-only
bandwidth	Integer	modifiable
axis	Rotation axis	read-only

compass : Compass

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

qt1 : Qt
qt2 : Qt
qt3 : Qt
qt4 : Qt

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

dataLogger : DataLogger

attribute	type	modifiable ?
logicalName	String	modifiable
advertisedValue	String	modifiable
currentRunIndex	Integer	read-only
timeUTC	UTC time	modifiable
recording	Enumerated	modifiable
autoStart	On/Off	modifiable
beaconDriven	On/Off	modifiable

usage	0..100%	read-only
clearHistory	Boolean	modifiable

6.3. Module

Global parameters control interface for all Yoctopuce devices

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

productName

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

serialNumber

Character string containing the serial number, unique and programmed at the factory. For a Yocto-3D-V2-C module, this serial number always starts with Y3DMK02C. You can use the serial number to access a given module by software.

logicalName

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

productId

USB device identifier of the module, preprogrammed to 208 at the factory.

productRelease

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

firmwareRelease

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

persistentSettings

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

luminosity

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

beacon

Activity of the localization beacon of the module.

upTime

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

usbCurrent

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

rebootCountdown

Countdown to use for triggering a reboot of the module.

userVar

32bit integer variable available for user storage.

6.4. Accelerometer

accelerometer control interface, available for instance in the Yocto-3D-V2 or the Yocto-Inclinometer

The `YAccelerometer` class allows you to read and configure Yoctopuce accelerometers. It inherits from `YSensor` class the core functions to read measurements, to register callback functions, and to access the autonomous datalogger. This class adds the possibility to access x, y and z components of the acceleration vector separately.

logicalName

Character string containing the logical name of the accelerometer, initially empty. This attribute can be modified at will by the user. Once initialized to a non-empty value, it can be used to access the accelerometer directly. If two accelerometers 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 accelerometer, that is automatically advertised up to the parent hub. For an accelerometer, the advertised value is the current value of the acceleration.

unit

Short character string representing the measuring unit for the acceleration.

currentValue

Current value of the acceleration, in g, as a floating point number.

lowestValue

Minimal value of the acceleration, in g, as a floating point number.

highestValue

Maximal value of the acceleration, in g, as a floating point number.

currentRawValue

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

logFrequency

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

reportFrequency

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

advMode

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

calibrationParam

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

resolution

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

sensorState

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

bandwidth

Measure update frequency.

xValue

X component of the acceleration, as a floating point number.

yValue

Y component of the acceleration, as a floating point number.

zValue

Z component of the acceleration, as a floating point number.

6.5. Magnetometer

magnetometer control interface, available for instance in the Yocto-3D-V2

The `YSensor` class is the parent class for all Yoctopuce sensor types. It can be used to read the current value and unit of any sensor, read the min/max value, configure autonomous recording frequency and access recorded data. It also provides a function to register a callback invoked each time the observed value changes, or at a predefined interval. Using this class rather than a specific subclass makes it possible to create generic applications that work with any Yoctopuce sensor, even those that do not yet exist. Note: The `YAnButton` class is the only analog input which does not inherit from `YSensor`.

logicalName

Character string containing the logical name of the magnetometer, initially empty. This attribute can be modified at will by the user. Once initialized to a non-empty value, it can be used to access the magnetometer directly. If two magnetometers 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 magnetometer, that is automatically advertised up to the parent hub. For a magnetometer, the advertised value is the current value of the magnetic field.

unit

Short character string representing the measuring unit for the magnetic field.

currentValue

Current value of the magnetic field, in mT, as a floating point number.

lowestValue

Minimal value of the magnetic field, in mT, as a floating point number.

highestValue

Maximal value of the magnetic field, in mT, as a floating point number.

currentRawValue

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

logFrequency

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

reportFrequency

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

advMode

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

calibrationParam

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

resolution

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

sensorState

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

bandwidth

Measure update frequency.

xValue

X component of the magnetic field, as a floating point number.

yValue

Y component of the magnetic field, as a floating point number.

zValue

Z component of the magnetic field, as a floating point number.

6.6. Gyro

gyroscope control interface, available for instance in the Yocto-3D-V2

The `YGyro` class allows you to read and configure Yoctopuce gyroscopes. It inherits from `YSensor` class the core functions to read measurements, to register callback functions, and to access the autonomous datalogger. This class adds the possibility to access x, y and z components of the rotation vector separately, as well as the possibility to deal with quaternion-based orientation estimates.

logicalName

Character string containing the logical name of the gyroscope, initially empty. This attribute can be modified at will by the user. Once initialized to a non-empty value, it can be used to access the gyroscope directly. If two gyroscopes 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 gyroscope, that is automatically advertised up to the parent hub. For a gyroscope, the advertised value is the current value of the angular velocity.

unit

Short character string representing the measuring unit for the angular velocity.

currentValue

Current value of the angular velocity, in degrees per second, as a floating point number.

lowestValue

Minimal value of the angular velocity, in degrees per second, as a floating point number.

highestValue

Maximal value of the angular velocity, in degrees per second, as a floating point number.

currentRawValue

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

logFrequency

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

reportFrequency

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

advMode

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

calibrationParam

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

resolution

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

sensorState

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

bandwidth

Measure update frequency.

xValue

Angular velocity around the X axis of the device, as a floating point number.

yValue

Angular velocity around the Y axis of the device, as a floating point number.

zValue

Angular velocity around the Z axis of the device, as a floating point number.

6.7. RefFrame

3D reference frame configuration interface, available for instance in the Yocto-3D-V2 or the Yocto-Inclinometer

The `YRefFrame` class is used to set up the base orientation of the Yoctopuce inertial sensors. Thanks to this, orientation functions relative to the earth surface plane can use the proper reference frame. For some devices, the class also implements a tridimensional sensor calibration process, which can compensate for local variations of standard gravity and improve the precision of the tilt sensors.

logicalName

Character string containing the logical name of the reference frame, initially empty. This attribute can be modified at will by the user. Once initialized to a non-empty value, it can be used to access the reference frame directly. If two reference frames 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 reference frame, that is automatically advertised up to the parent hub. For a reference frame, the advertised value is its the ordered list of axis device corresponding to roll, pitch and heading.

mountPos

Installation position of the device, defining the reference frame for the compass and the pitch/roll tilt sensors.

bearing

Reference bearing used by the compass (if present).

calibrationParam

Low-level calibration parameters for the accelerometer and magnetometer used by the compass and pitch/roll tilt sensors (when supported).

fusionMode

Fusion algorithm used to combine the accelerometer, gyroscope and compass data (when supported).

6.8. Tilt

tilt sensor control interface, available for instance in the Yocto-3D-V2 or the Yocto-Inclinometer

The `YSensor` class is the parent class for all Yoctopuce sensor types. It can be used to read the current value and unit of any sensor, read the min/max value, configure autonomous recording frequency and access recorded data. It also provides a function to register a callback invoked each

time the observed value changes, or at a predefined interval. Using this class rather than a specific subclass makes it possible to create generic applications that work with any Yoctopuce sensor, even those that do not yet exist. Note: The `YAnButton` class is the only analog input which does not inherit from `YSensor`.

logicalName

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

advertisedValue

Short character string summarizing the current state of the tilt sensor, that is automatically advertised up to the parent hub. For a tilt sensor, the advertised value is the current value of the inclination.

unit

Short character string representing the measuring unit for the inclination.

currentValue

Current value of the inclination, in degrees, as a floating point number.

lowestValue

Minimal value of the inclination, in degrees, as a floating point number.

highestValue

Maximal value of the inclination, in degrees, as a floating point number.

currentRawValue

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

logFrequency

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

reportFrequency

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

advMode

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

calibrationParam

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

resolution

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

sensorState

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

bandwidth

Measure update frequency.

axis

Function working axis for the tilt sensor.

6.9. Compass

compass function control interface, available for instance in the Yocto-3D-V2

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

logicalName

Character string containing the logical name of the compass function, initially empty. This attribute can be modified at will by the user. Once initialized to a non-empty value, it can be used to access the compass function directly. If two compass functions 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 compass function, that is automatically advertised up to the parent hub. For a compass function, the advertised value is the current value of the relative bearing.

unit

Short character string representing the measuring unit for the relative bearing.

currentValue

Current value of the relative bearing, in degrees, as a floating point number.

lowestValue

Minimal value of the relative bearing, in degrees, as a floating point number.

highestValue

Maximal value of the relative bearing, in degrees, as a floating point number.

currentRawValue

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

logFrequency

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

reportFrequency

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

advMode

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

calibrationParam

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

resolution

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

sensorState

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

bandwidth

Measure update frequency.

axis

Function working axis for the compass function.

magneticHeading

Magnetic heading (regardless of the configured bearing)

6.10. DataLogger

DataLogger control interface, available on most Yoctopuce sensors.

A non-volatile memory for storing ongoing measured data is available on most Yoctopuce sensors. Recording can happen automatically, without requiring a permanent connection to a computer. The `YDataLogger` class controls the global parameters of the internal data logger. Recording control (start/stop) as well as data retrieval is done at sensor objects level.

logicalName

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

advertisedValue

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

currentRunIndex

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

timeUTC

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

recording

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

autoStart

Automatic start of the data logger on power on. Setting this attribute ensures that the data logger is always turned on when the device is powered up, without need for a software command. Note: if the device doesn't have any time source at his disposal, it will wait for ~8 seconds before automatically starting to record.

beaconDriven

Synchronize the state of the localization beacon and the state of the data logger. If this attribute is set, it is possible to start the recording with the Yocto-button or the attribute `beacon` of the function `YModule`. In the same way, if the attribute `recording` is changed, the state of the localization beacon is updated. Note: when this attribute is set the localization beacon pulses at a slower rate than usual.

usage

Percentage of datalogger memory in use.

clearHistory

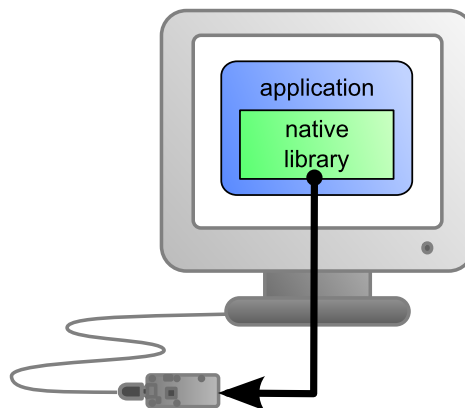
Attribute that can be set to true to clear recorded data.

6.11. What interface: Native, DLL or Service ?

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

Native control

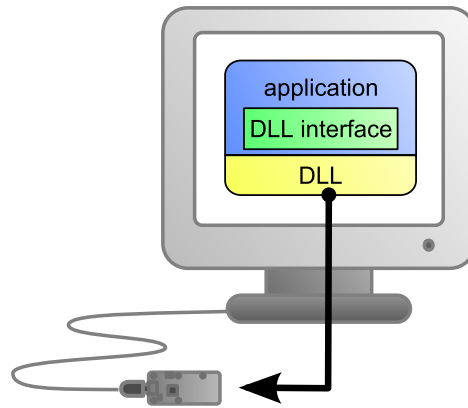
In this case, the software driving your project is compiled directly with a library which provides control of the modules. Objectively, it is the simplest and most elegant solution for the end user. The end user then only needs to plug the USB cable and run your software for everything to work. Unfortunately,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



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

Native control by DLL

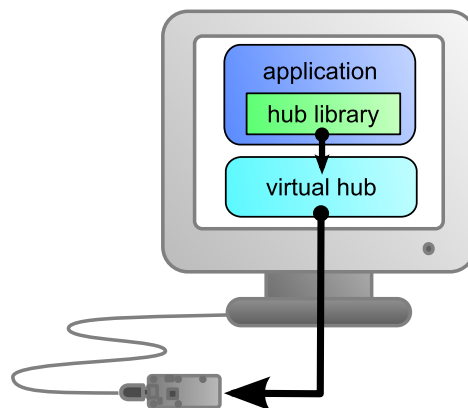
Here, the main part of the code controlling the modules is located in a DLL. The software is compiled with a small library which provides control of the DLL. It is the fastest method to code module support in a given language. Indeed, the "useful" part of the control code is located in the DLL which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



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

Control by service

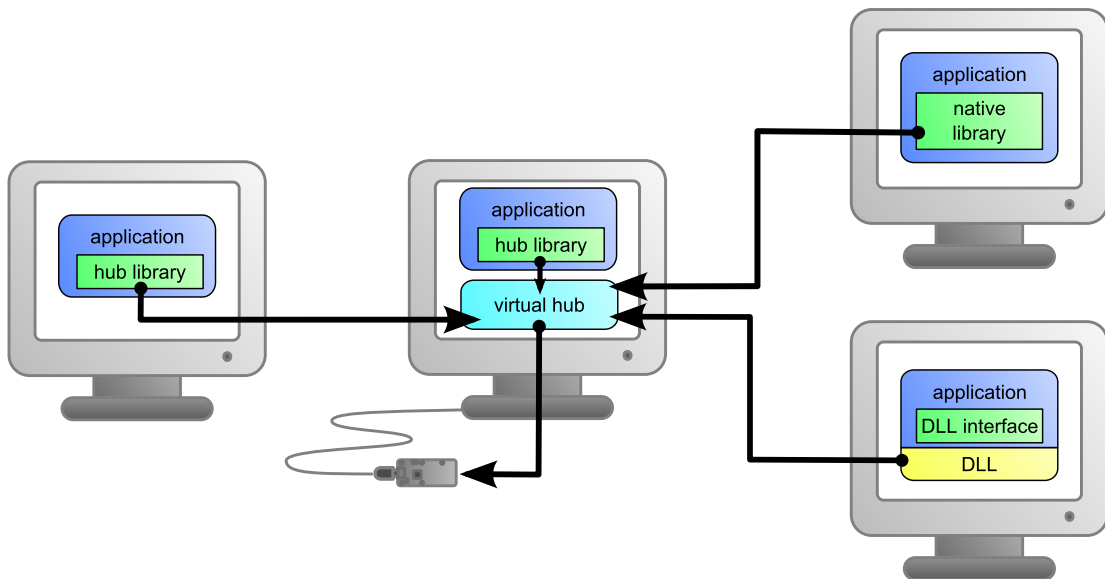
Some languages do simply not allow you to easily gain access to the hardware layers of the machine. It is the case for Javascript, for instance. To deal with this case, Yoctopuce provides a solution in the form of a small piece of software called *VirtualHub*¹. It can access the modules, and your application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



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

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

¹ www.yoctopuce.com/EN/virtualhub.php



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

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

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

Support methods for different languages

6.12. Accessing modules through a hub

VirtualHub to work around USB access limitation

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

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

Another process is already using yAPI

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

With a YoctoHub

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

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

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

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

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

6.13. Programming, where to start?

At this point of the user's guide, you should know the main theoretical points of your Yocto-3D-V2-C. It is now time to practice. You must download the Yoctopuce library for your favorite programming language from the Yoctopuce web site². Then skip directly to the chapter corresponding to the chosen programming language.

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

When you have mastered the basic programming of your module, you can turn to the chapter on advanced programming that describes some techniques that will help you make the most of your Yocto-3D-V2-C.

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

7. Using the Yocto-3D-V2-C in command line

When you want to perform a punctual operation on your Yocto-3D-V2-C, such as reading a value, assigning a logical name, and so on, you can obviously use VirtualHub, but there is a simpler, faster, and more efficient method: the command line API.

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

7.1. Installing

Download the command line API². You do not need to run any setup, simply copy the executables corresponding to your platform/OS in a directory of your choice. You may add this directory to your PATH variable to be able to access these executables from anywhere. You are all set, you only need to connect your Yocto-3D-V2-C, open a shell, and start working by typing for example:

```
C:\>YTilt any get_currentValue
```

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

7.2. Use: general description

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

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

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

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

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

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

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

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

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

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

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

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

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

7.3. Control of the Tilt function

To control the Tilt function of your Yocto-3D-V2-C, you need the YTilt executable file.

For instance, you can launch:

```
C:\>YTilt any get_currentValue
```

This example uses the "any" target to indicate that we want to work on the first Tilt 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-3D-V2-C module with the Y3DMK02C-123456 serial number which you have called "MyModule", and its tilt1 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:\>YTilt Y3DMK02C-123456.tilt1 describe
C:\>YTilt Y3DMK02C-123456.MyFunction describe
C:\>YTilt MyModule.tilt1 describe
C:\>YTilt MyModule.MyFunction describe
C:\>YTilt MyFunction describe
```

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

```
C:\>YTilt all describe
```

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

```
C:\>YTilt /help
```

7.4. Control of the module part

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

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

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

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

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

```
C:\>YModule Y3DMK02C-12346 set_logicalName MonPremierModule
C:\>YModule Y3DMK02C-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 Y3DMK02C-12346 set_logicalName MonPremierModule
C:\>YModule Y3DMK02C-12346 saveToFlash
```

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

```
C:\>YModule -s Y3DMK02C-12346 set_logicalName MonPremierModule
```

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

7.5. Limitations

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

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

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

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

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

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

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

8. Using the Yocto-3D-V2-C with Python

Python is an interpreted object oriented language developed by Guido van Rossum. Among its advantages is the fact that it is free, and the fact that it is available for most platforms, Windows as well as UNIX. It is an ideal language to write small scripts on a napkin. The Yoctopuce library is compatible with Python 2.7 and 3.x up to the latest official versions. It works under Windows, macOS, and Linux, Intel as well as ARM. Python interpreters are available on the Python web site¹.

8.1. Source files

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

8.2. Dynamic library

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

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

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

8.3. Control of the Tilt function

A few lines of code are enough to use a Yocto-3D-V2-C. Here is the skeleton of a Python code snippet to use the Tilt function.

¹ <http://www.python.org/download/>

² www.yoctopuce.com/EN/libraries.php

```
[...]
# Enable detection of USB devices
errmsg=YRefParam()
YAPI.RegisterHub("usb",errmsg)
[...]

# Retrieve the object used to interact with the device
tilt = YTilt.FindTilt("Y3DMK02C-123456.tilt1")

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

[...]
```

Let's look at these lines in more details.

YAPI.RegisterHub

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

YTilt.FindTilt

The `YTilt.FindTilt` function allows you to find a tilt sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-3D-V2-C module with serial number `Y3DMK02C-123456` which you have named "MyModule", and for which you have given the `tilt1` function the name "MyFunction". The following five calls are strictly equivalent, as long as "MyFunction" is defined only once.

```
tilt = YTilt.FindTilt("Y3DMK02C-123456.tilt1")
tilt = YTilt.FindTilt("Y3DMK02C-123456.MyFunction")
tilt = YTilt.FindTilt("MyModule.tilt1")
tilt = YTilt.FindTilt("MyModule.MyFunction")
tilt = YTilt.FindTilt("MyFunction")
```

`YTilt.FindTilt` returns an object which you can then use at will to control the tilt sensor.

isOnline

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

About python imports

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

```
from yocto_api import *
```

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

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

get_currentValue

The `get_currentValue()` method of the object returned by `YTilt.FindTilt` provides the value of orientation measured by the tilt sensor. The value returned is a floating number.

YCompass.FindCompass, YGyro.FindGyro...

Functions `YCompass.FindCompass`, `YMagnetometer.FindMagnetometer`, `YGyro.FindGyro` and `YAccelerometer.FindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `YTilt.FindTilt`.

A real example

Launch Python and open the corresponding sample script provided in the directory **Examples/Doc-GettingStarted-Yocto-3D-V2-C** of the Yoctopuce library.

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

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

from yocto_api import *
from yocto_tilt import *
from yocto_compass import *
from yocto_gyro import *
from yocto_accelerometer import *

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

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

errmsg = YRefParam()

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

target = sys.argv[1]

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

if target == 'any':
    # retrieve any tilt sensor
    anytilt = YTilt.FirstTilt()
    if anytilt is None:
        die('No module connected (check USB cable)')
    m = anytilt.get_module()
    target = m.get_serialNumber()
else:
    anytilt = YTilt.FindTilt(target + ".tilt1")
    if not (anytilt.isOnline()):
        die('Module not connected (check identification and USB cable)')

serial = anytilt.get_module().get_serialNumber()
tilt1 = YTilt.FindTilt(serial + ".tilt1")
tilt2 = YTilt.FindTilt(serial + ".tilt2")
compass = YCompass.FindCompass(serial + ".compass")
accelerometer = YAccelerometer.FindAccelerometer(serial + ".accelerometer")
gyro = YGyro.FindGyro(serial + ".gyro")

count = 0

if not (tilt1.isOnline()):
    die("Module not connected (check identification and USB cable)")

while tilt1.isOnline():
```

```

if count % 10 == 0:
    print("tilt1    tilt2    compass acc    gyro")

    print("%-7.1f " % tilt1.get_currentValue() + \
          "%-7.1f " % tilt2.get_currentValue() + \
          "%-7.1f " % compass.get_currentValue() + \
          "%-7.1f " % accelerometer.get_currentValue() + \
          "%-7.1f" % gyro.get_currentValue())
    count += 1
    YAPI.Sleep(250, errmsg)
YAPI.FreeAPI()

```

8.4. Control of the module part

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

```

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

from yocto_api import *

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

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

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

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

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

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

```

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

Changing the module settings

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

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

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

from yocto_api import *

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

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

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

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

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

Listing the modules

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

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

from yocto_api import *

errmsg = YRefParam()

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

print('Device list')

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

8.5. Error handling

When you implement a program which must interact with USB modules, you cannot disregard error handling. Inevitably, there will be a time when a user will have unplugged the device, either before

running the software, or even while the software is running. The Yoctopuce library is designed to help you support this kind of behavior, but your code must nevertheless be conceived to interpret in the best possible way the errors indicated by the library.

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

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

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

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

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

9. Using Yocto-3D-V2-C with C++

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

Yoctopuce C++ libraries¹ are integrally provided as source files. A section of the low-level library is written in pure C, but you should not need to interact directly with it: everything was done to ensure the simplest possible interaction from C++. The library is naturally also available as binary files, so that you can link it directly if you prefer.

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

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

9.1. Control of the Tilt function

A few lines of code are enough to use a Yocto-3D-V2-C. Here is the skeleton of a C++ code snippet to use the Tilt function.

```
#include "yocto_api.h"
#include "yocto_tilt.h"

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

// Retrieve the object used to interact with the device
YTilt *tilt;
```

¹ www.yoctopuce.com/EN/libraries.php

```

tilt = YTilt::FindTilt("Y3DMK02C-123456.tilt1");

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

```

Let's look at these lines in more details.

yocto_api.h et yocto_tilt.h

These two include files provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api.h` must always be used, `yocto_tilt.h` is necessary to manage modules containing a tilt sensor, such as Yocto-3D-V2-C.

YAPI::RegisterHub

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

YTilt::FindTilt

The `YTilt::FindTilt` function allows you to find a tilt sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-3D-V2-C module with serial number `Y3DMK02C-123456` which you have named "MyModule", and for which you have given the `tilt1` function the name "MyFunction". The following five calls are strictly equivalent, as long as "MyFunction" is defined only once.

```

YTilt *tilt = YTilt::FindTilt("Y3DMK02C-123456.tilt1");
YTilt *tilt = YTilt::FindTilt("Y3DMK02C-123456.MyFunction");
YTilt *tilt = YTilt::FindTilt("MyModule.tilt1");
YTilt *tilt = YTilt::FindTilt("MyModule.MyFunction");
YTilt *tilt = YTilt::FindTilt("MyFunction");

```

`YTilt::FindTilt` returns an object which you can then use at will to control the tilt sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `yFindTilt` provides the angle currently measured by the inclinometer. The value returned is a floating number.

yFindCompass and yFindGyro

Functions `yFindCompass`, `yFindGyro`, `yFindMagnetometer` and `yFindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `yFindTilt`.

A real example

Launch your C++ environment and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-3D-V2-C** of the Yoctopuce library. If you prefer to work with your favorite text editor, open the file `main.cpp`, and type `make` to build the example when you are done.

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

```

#include "yocto_api.h"
#include "yocto_tilt.h"
#include "yocto_compass.h"
#include "yocto_gyro.h"
#include "yocto_accelerometer.h"
#include <iostream>
#include <stdlib.h>
#include <iostream>
#include <iomanip>

using namespace std;

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

int main(int argc, const char * argv[])
{
    string errmsg, target;
    YTilt *anytilt, *tilt1, *tilt2;
    YCompass *compass;
    YAccelerometer *accelerometer;
    YGyro *gyro;

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

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

    // try to find a valid serial number
    if(target == "any") {
        anytilt = YTilt::FirstTilt();
        if (anytilt == NULL) {
            cout << "No module connected (check USB cable)" << endl;
            return 1;
        }
    } else {
        anytilt = YTilt::FindTilt(target + ".tilt1");
        if (!anytilt->isOnline()) {
            cout << "Module not connected (check identification and USB cable)" << endl;
            return 1;
        }
    }
    string serial = anytilt->get_module()->get_serialNumber();

    // retrieve all sensors on the device matching the serial
    tilt1 = YTilt::FindTilt(serial + ".tilt1");
    tilt2 = YTilt::FindTilt(serial + ".tilt2");
    compass = YCompass::FindCompass(serial + ".compass");
    accelerometer = YAccelerometer::FindAccelerometer(serial + ".accelerometer");
    gyro = YGyro::FindGyro(serial + ".gyro");
    int count = 0;

    while(1) {
        if(!tilt1->isOnline()) {
            cout << "device disconnected";
            break;
        }
        if ((count % 10) == 0) {
            cout << "tilt1\ntilt2\ncompass\tacc\tgyro" << endl;
        }
        cout << std::setprecision(2) << std::setw(8)
            << tilt1->get_currentValue() << "\t"
            << tilt2->get_currentValue() << "\t"

```

```

        << compass->get_currentValue() << "\t"
        << accelerometer->get_currentValue() << "\t"
        << gyro->get_currentValue() << endl;
    count++;
    YAPI::Sleep(250, errmsg);
}
YAPI::FreeAPI();
return 0;
}

```

9.2. Control of the module part

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

```

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

#include "yocto_api.h"

using namespace std;

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

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

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

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

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

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

```

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

Changing the module settings

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

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

#include "yocto_api.h"

using namespace std;

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

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

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

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

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

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

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

Listing the modules

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

```
#include <iostream>
```

```

#include "yocto_api.h"

using namespace std;

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

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

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

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

```

9.3. Error handling

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

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

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

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

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

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

9.4. Integration variants for the C++ Yoctopuce library

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

Integration in source format (recommended)

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

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

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

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

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

Integration as a static library

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

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

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

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

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

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

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

Integration as a dynamic library

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

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

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

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

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

With GCC, the command line to compile is simply:

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

10. Using Yocto-3D-V2-C with C#

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

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

10.1. Installation

Download the Visual C# Yoctopuce library from the Yoctopuce web site¹. There is no setup program, simply copy the content of the zip file into the directory of your choice. You mostly need the content of the `Sources` directory. The other directories contain the documentation and a few sample programs. All sample projects are Visual C# 2010, projects, if you are using a previous version, you may have to recreate the projects structure from scratch.

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

The Visual C#.NET Yoctopuce library is composed of a DLL and of source files in Visual C#. The DLL is not a .NET DLL, but a classic DLL, written in C, which manages the low level communications with the modules². The source files in Visual C# manage the high level part of the API. Therefore, you need both this DLL and the .cs files of the `sources` directory to create a project managing Yoctopuce modules.

Configuring a Visual C# project

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

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

¹ www.yoctopuce.com/EN/libraries.php

² The sources of this DLL are available in the C++ API

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

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

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

10.3. Control of the Tilt function

A few lines of code are enough to use a Yocto-3D-V2-C. Here is the skeleton of a C# code snippet to use the Tilt function.

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

// Retrieve the object used to interact with the device
YTilt tilt = YTilt.FindTilt("Y3DMK02C-123456.tilt1");

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

Let's look at these lines in more details.

YAPI.RegisterHub

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

YTilt.FindTilt

The `YTilt.FindTilt` function allows you to find a tilt sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-3D-V2-C module with serial number `Y3DMK02C-123456` which you have named `"MyModule"`, and for which you have given the `tilt1` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
tilt = YTilt.FindTilt("Y3DMK02C-123456.tilt1");
tilt = YTilt.FindTilt("Y3DMK02C-123456.MyFunction");
tilt = YTilt.FindTilt("MyModule.tilt1");
tilt = YTilt.FindTilt("MyModule.MyFunction");
tilt = YTilt.FindTilt("MyFunction");
```

`YTilt.FindTilt` returns an object which you can then use at will to control the tilt sensor.

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

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `YTilt.FindTilt` provides the value of orientation measured by the tilt sensor. The value returned is a floating number.

YCompass.FindCompass, YGyro.FindGyro...

Functions `YCompass.FindCompass`, `YMagnetometer.FindMagnetometer`, `YGyro.FindGyro` and `YAccelerometer.FindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `YTilt.FindTilt`.

A real example

Launch Microsoft Visual C# and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-3D-V2-C** of the Yoctopuce library.

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

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

namespace ConsoleApplication1
{
    class Program
    {
        static void 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 errormsg = "";
            string target;

            YTilt anytilt, tilt1, tilt2;
            YCompass compass;
            YAccelerometer accelerometer;
            YGyro gyro;

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

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

            if (target == "ANY") {
                anytilt = YTilt.FirstTilt();
                if (anytilt == null) {
                    Console.WriteLine("No module connected (check USB cable)");
                    Environment.Exit(0);
                }
            } else {
                anytilt = YTilt.FindTilt(target + ".tilt1");
                if (!anytilt.isOnline()) {
```

```

        Console.WriteLine("Module not connected");
        Console.WriteLine("check identification and USB cable");
        Environment.Exit(0);
    }
}

string serial = anytilt.get_module().get_serialNumber();
tilt1 = YTilt.FindTilt(serial + ".tilt1");
tilt2 = YTilt.FindTilt(serial + ".tilt2");
compass = YCompass.FindCompass(serial + ".compass");
accelerometer = YAccelerometer.FindAccelerometer(serial + ".accelerometer");
gyro = YGyro.FindGyro(serial + ".gyro");
int count = 0;

if (!tilt1.isOnline()) {
    Console.WriteLine("device disconnected");
    Environment.Exit(0);
}

while (tilt1.isOnline()) {

    if (count % 10 == 0) Console.WriteLine("tilt1    tilt2    compass    acc    gyro");

    Console.Write(tilt1.get_currentValue().ToString() + "\t");
    Console.Write(tilt2.get_currentValue().ToString() + "\t");
    Console.Write(compass.get_currentValue().ToString() + "\t");
    Console.Write(accelerometer.get_currentValue().ToString() + "\t");
    Console.WriteLine(gyro.get_currentValue().ToString());

    YAPI.Sleep(250, ref errmsg);
}
YAPI.FreeAPI();
}
}
}

```

10.4. Control of the module part

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

```

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

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

        static void Main(string[] args)
        {
            YModule m;
            string 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.WriteLine("beacon:     ");
            if (m.get_beacon() == YModule.BEACON_ON)
                Console.WriteLine("ON");
            else
                Console.WriteLine("OFF");
            Console.WriteLine("upTime:      " + (m.get_upTime() / 1000 ).ToString() + " sec");
            Console.WriteLine("USB current: " + m.get_usbCurrent().ToString() + " mA");
            Console.WriteLine("Logs:\x\n" + m.get_lastLogs());

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

```

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

Changing the module settings

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

```

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

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

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

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

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

```

```

    }

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

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

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

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

```

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

Listing the modules

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

```

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

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

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

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

```

10.5. Error handling

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

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

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

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

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

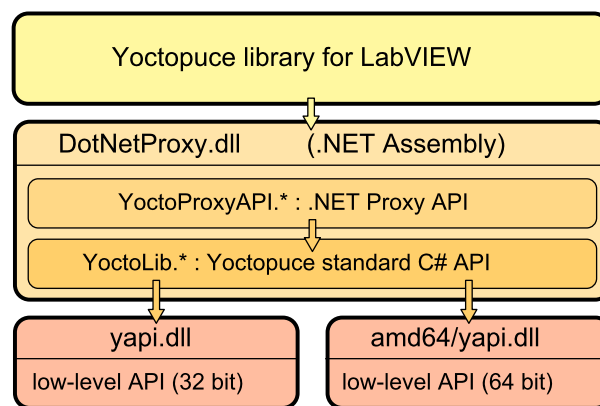
When you work without exceptions, you can obtain an error code and an error message explaining the source of the error. You can request them from the object which returned the error, calling the `errType()` and `errMessage()` methods. Their returned values contain the same information as in the exceptions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

11. Using the Yocto-3D-V2-C with LabVIEW

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

11.1. Architecture

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



LabVIEW Yoctopuce API architecture

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

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

11.2. Compatibility

Firmware

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

LabVIEW for Linux and MacOS

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

LabVIEW NXG

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

About DotNetProxyLibrary.dll

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

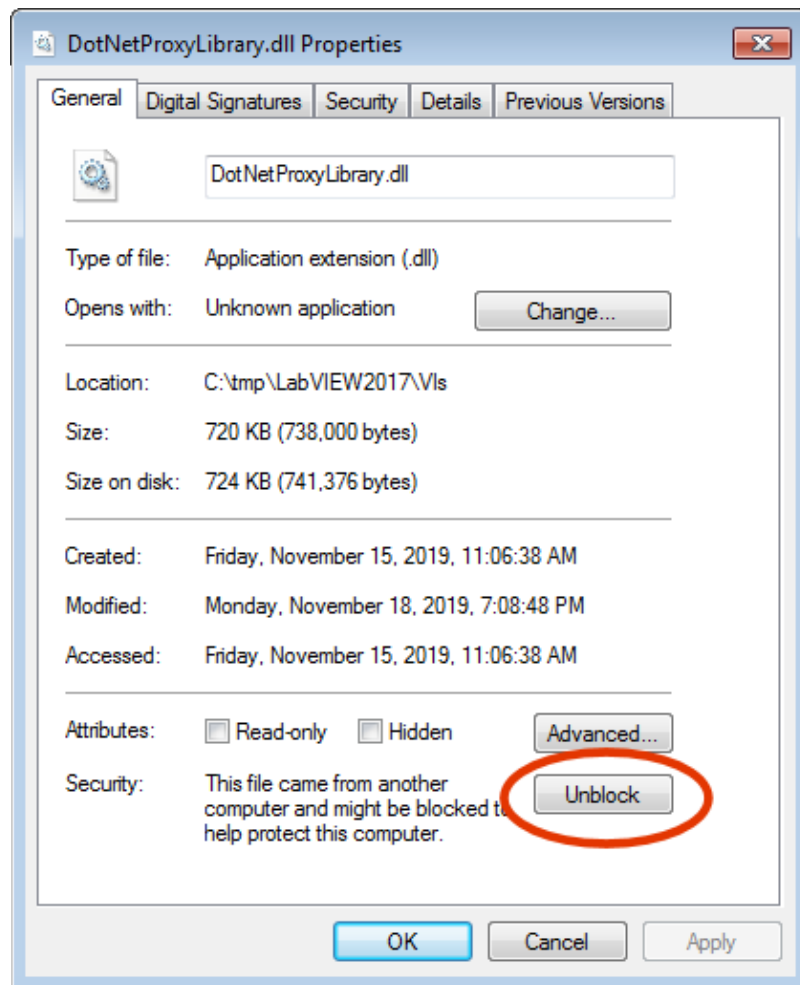
11.3. Installation

Download the LabVIEW library from the Yoctopuce web site¹. It is a ZIP file in which there is a distinct directory for each version of LabVIEW. Each of these directories contains two subdirectories: the first one contains programming examples for each Yoctopuce product; the second one, called *VIs*, contains all the VIs of the API and the required DLLs.

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

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

¹ <http://www.yoctopuce.com/EN/libraries.php>



Unblock the DotNetProxyLibrary DLL.

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

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

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

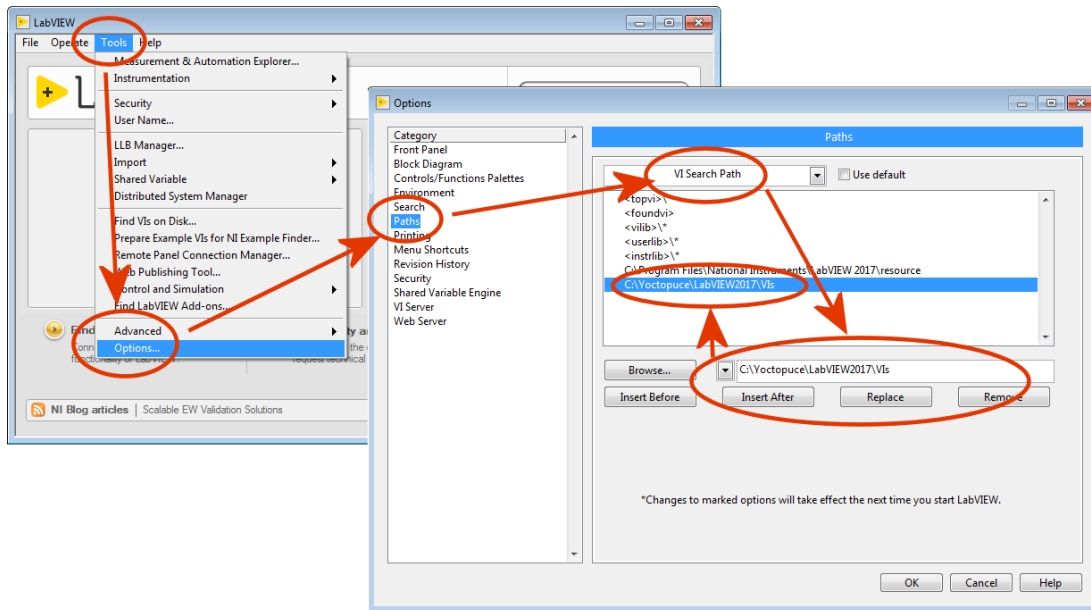
To install the LabVIEW Yoctopuce API, there are several methods.

Method 1 : "Take-out" installation

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

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

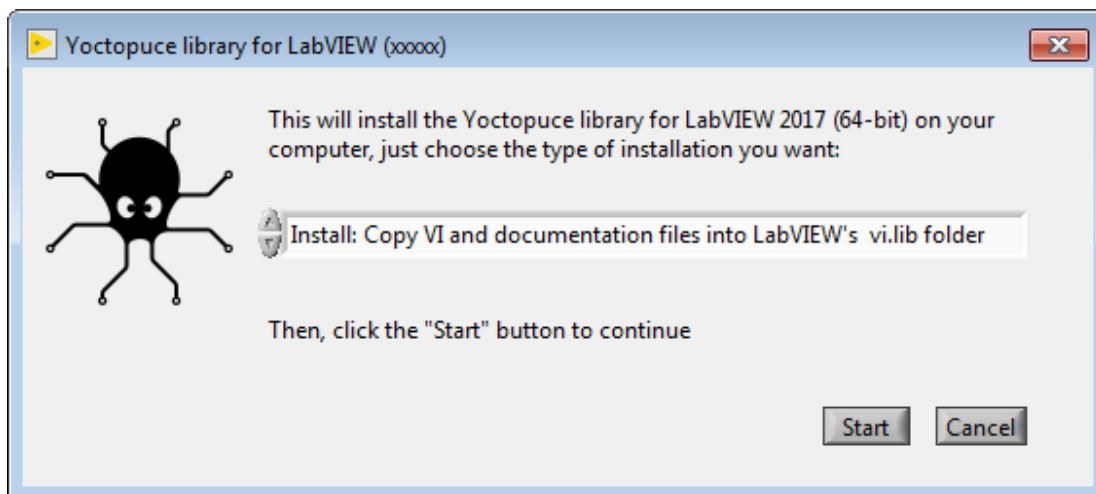
² <https://knowledge.ni.com/KnowledgeArticleDetails?id=kA00Z000000P8XnSAK>



Configuring the "VI Search Path"

Method 2 : Provided installer

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



The provider installer

This installer provide 3 installation options:

Install: Keep VI and documentation files where they are.

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

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

Install: Copy VI and documentation files into LabVIEW's vi.lib folder

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

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

Uninstall Yoctopuce Library

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

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

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

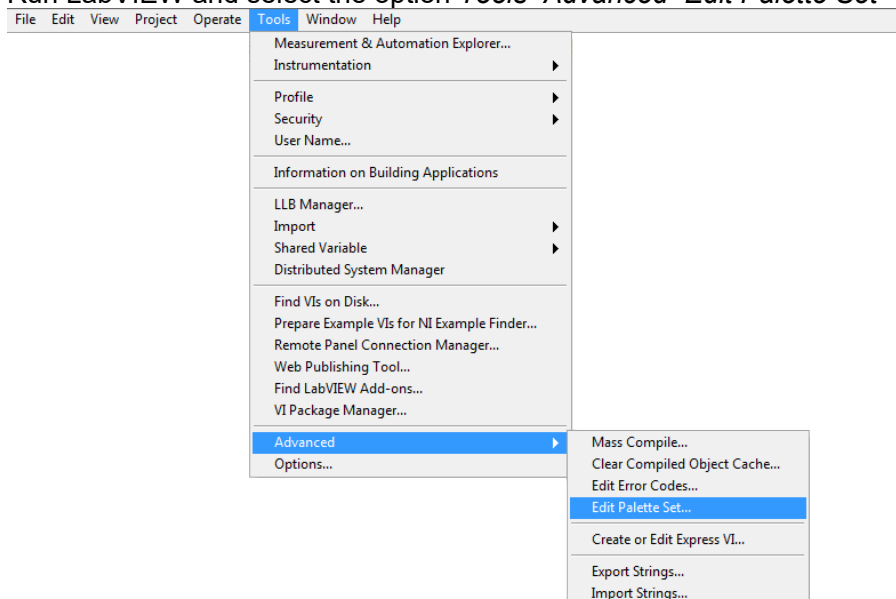
The installer identifies Yoctopuce VIs library folders by checking the presence of the *YRegisterHub.vi* file in said folders.

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

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

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

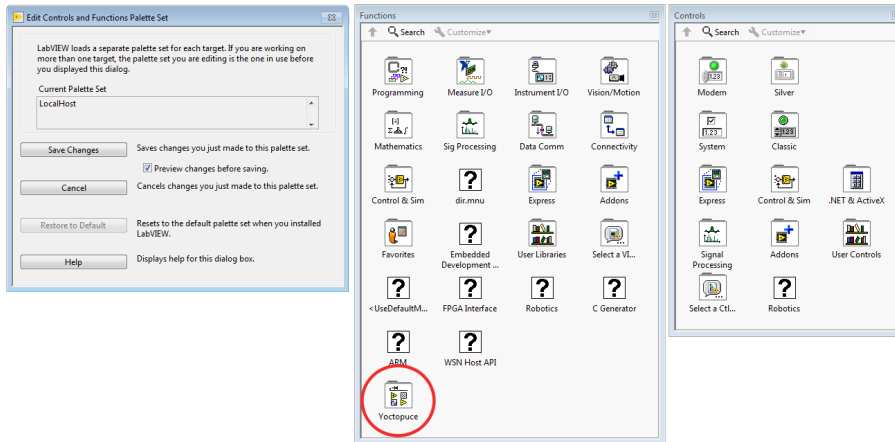
1. Create a *Yoctopuce/API* directory in the *C:\Program Files\National Instruments\LabVIEW xxxx\vi.lib* directory and copy all the VIs and DLLs of the *VIs* directory into it.
2. Create a *Yoctopuce* directory in the *C:\Program Files\National Instruments\LabVIEW xxxx\menus\Categories* directory.
3.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



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

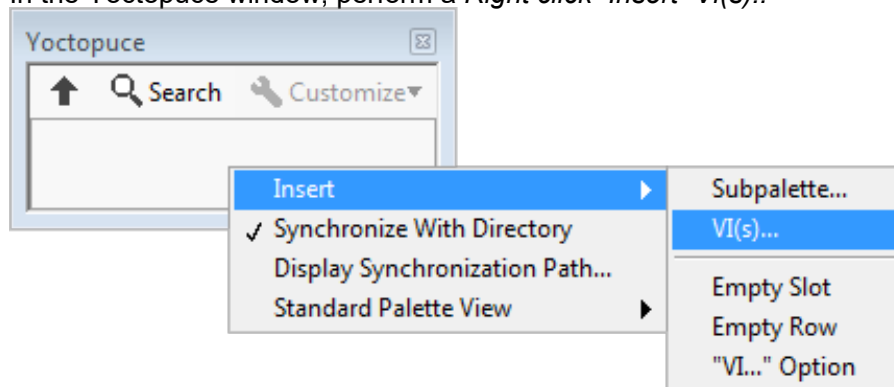
Three windows pop up:

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

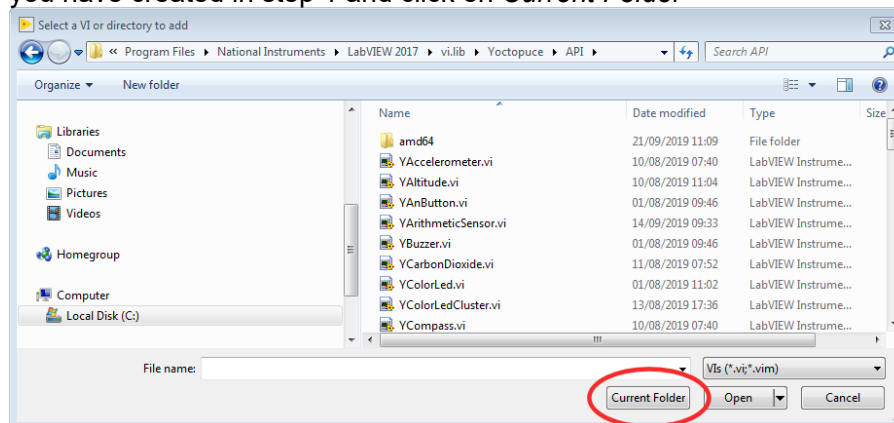


In the *Function* window, there is a *Yoctopuce* icon. Double-click it to create an empty "Yoctopuce" window.

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

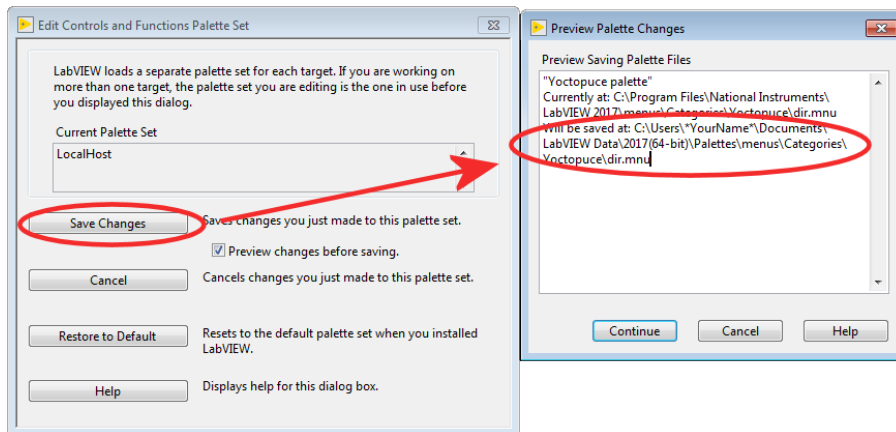


in order to open a file chooser. Put the file chooser in the *vi.lib\Yoctopuce\API* directory that you have created in step 1 and click on *Current Folder*



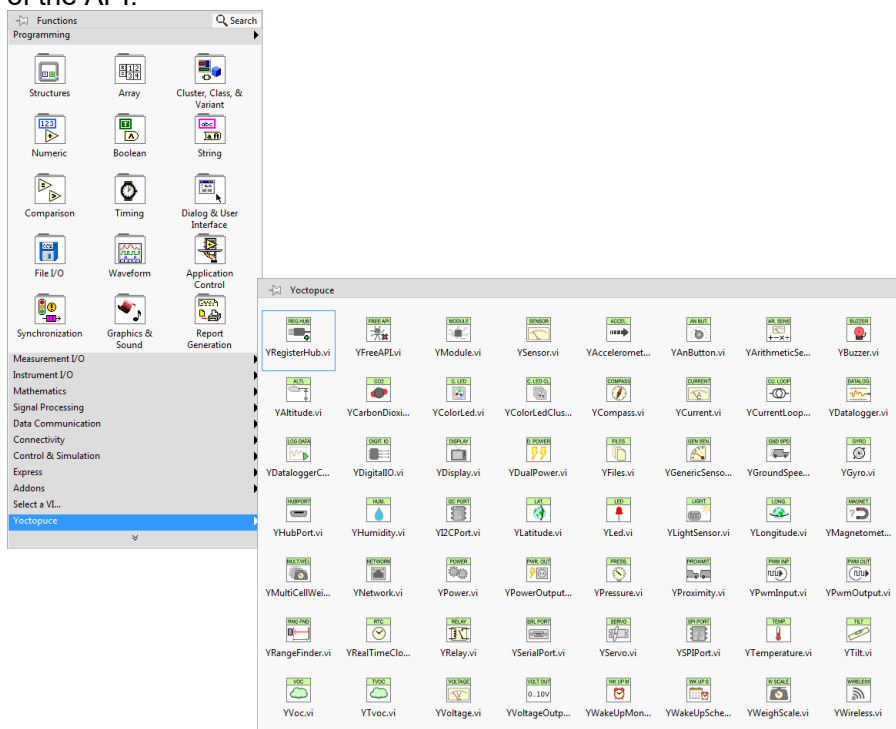
All the Yoctopuce VIs now appear in the Yoctopuce window. By default, they are sorted by alphabetical order, but you can arrange them as you see fit by moving them around with the mouse. For the palette to be easy to use, we recommend to reorganize the icons over 8 columns.

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



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

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

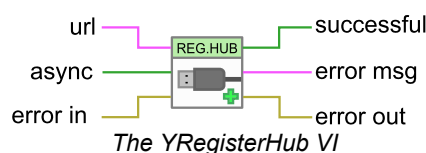


11.4. Presentation of Yoctopuce VIs

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

YRegisterHub

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



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

- The "usb" character string to indicated that you wish to work with local modules, directly connected by USB
- An IP address to indicate that you wish to work with modules which are available through a network connection. This IP address can be that of a YoctoHub⁴ or even that of a machine on which the VirtualHub⁵ application is running.

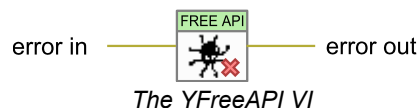
In the case of an IP address, the YRegisterHub VI tries to contact this address and generates an error if it does not succeed, unless the *async* parameter is set to TRUE. If *async* is set to TRUE, no error is generated and Yoctopuce modules corresponding to that IP address become automatically available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

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

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

YFreeAPI

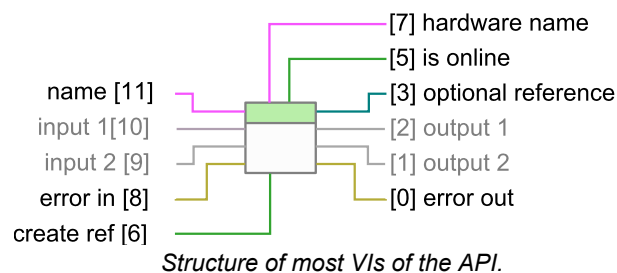
The YFreeAPI VI enables you to free resources allocated by the Yoctopuce API.



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

Structure of the VIs corresponding to a class

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



- Connector [11]: *name* must contain the hardware name or the logical name of the intended function.
- Connectors [10] and [9]: input parameters depending on the nature of the VI.
- Connectors [8] and [0] : *error in* and *error out*.
- Connector [7] : Unique hardware name of the found function.
- Connector [5] : *is online* contains TRUE if the function is available, FALSE otherwise.
- Connectors [2] and [1]: output values depending on the nature of the VI.
- Connector [6]: If this input is set to TRUE, connector [3] contains a reference to the *Proxy* objects implemented by the VI⁶. This input is initialized to FALSE by default.

⁴ www.yoctopuce.com/EN/products/category/extensions-and-networking

⁵ <http://www.yoctopuce.com/EN/virtualhub.php>

⁶ see section *Using Proxy objects*

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

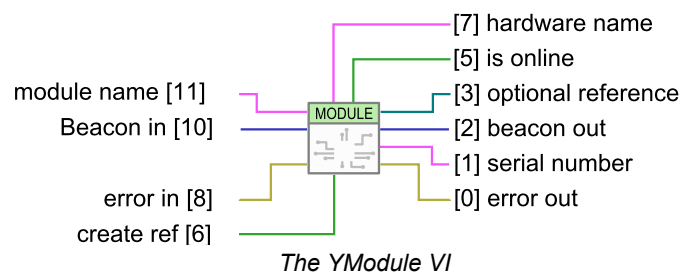
You can find the list of functions available on your Yocto-3D-V2-C in chapter *Programming, general concepts*.

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

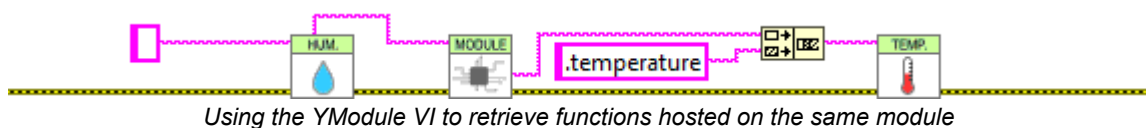
If the *name* parameter contains an empty string, the VI targets the first available function of the same type. If no function is available, *is online* is set to FALSE.

The YModule VI

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

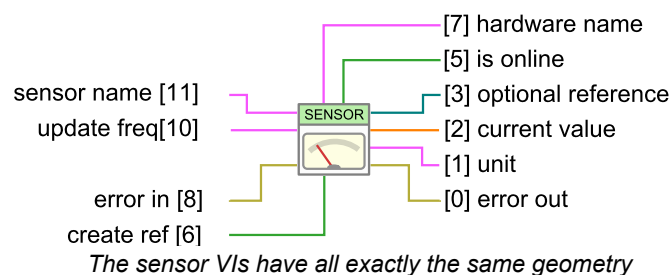


The *name* input works slightly differently from other VIs. If it is called with a *name* parameter corresponding to a function name, the `YModule` VI finds the *Module* function of the module hosting the function. You can therefore easily find the serial number of the module of any function. This enables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



The sensor VIs

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

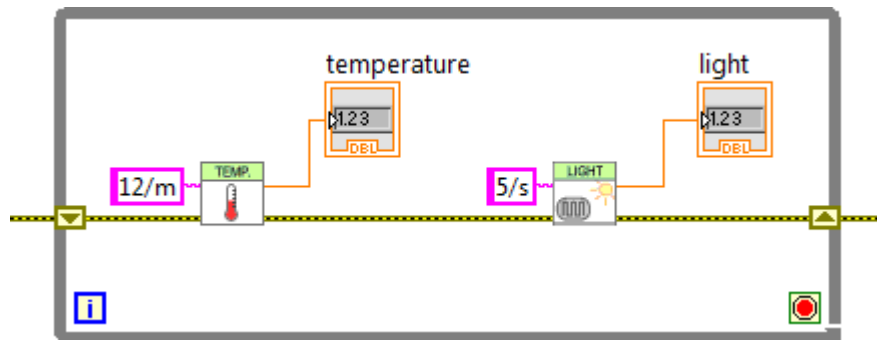


The *update freq* input parameter is a character string enabling you to configure the way in which the output value is updated:

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

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

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

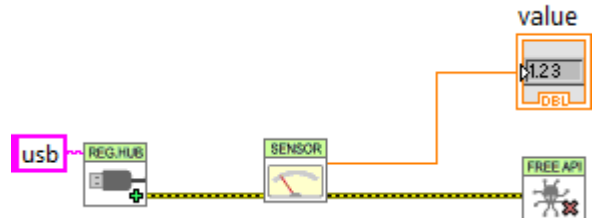


Changing the update frequency of the same module

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

11.5. Functioning and use of VIs

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

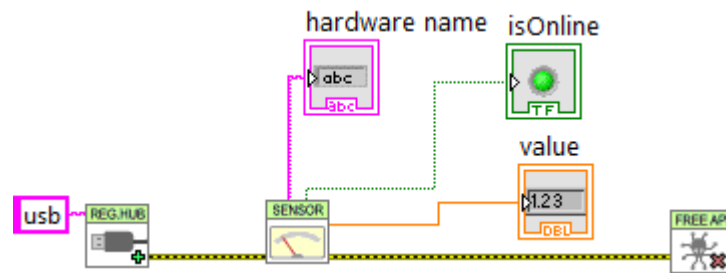


Minimal example of use of the LabVIEW Yoctopuce API

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

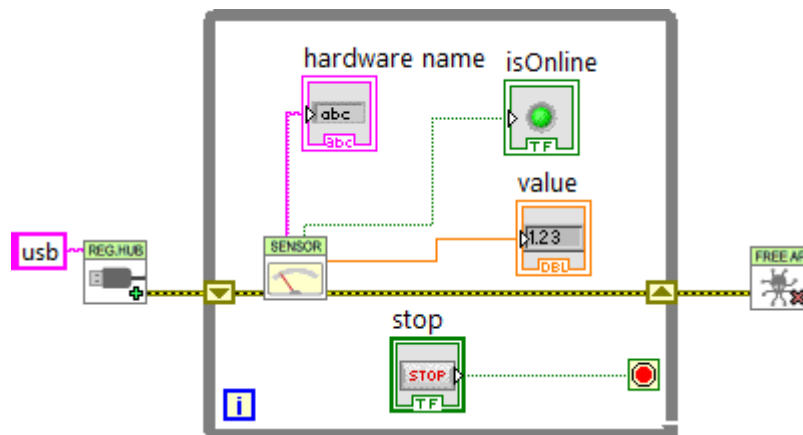
1. It initializes the API in native ("usb") mode with the `YRegisterHub` VI.
2. It displays the value of the first Yoctopuce sensor it finds thanks to the `YSensor` VI.
3. It frees the API thanks to the `YFreeAPI` VI.

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



Use of the hardware name and isOnline outputs

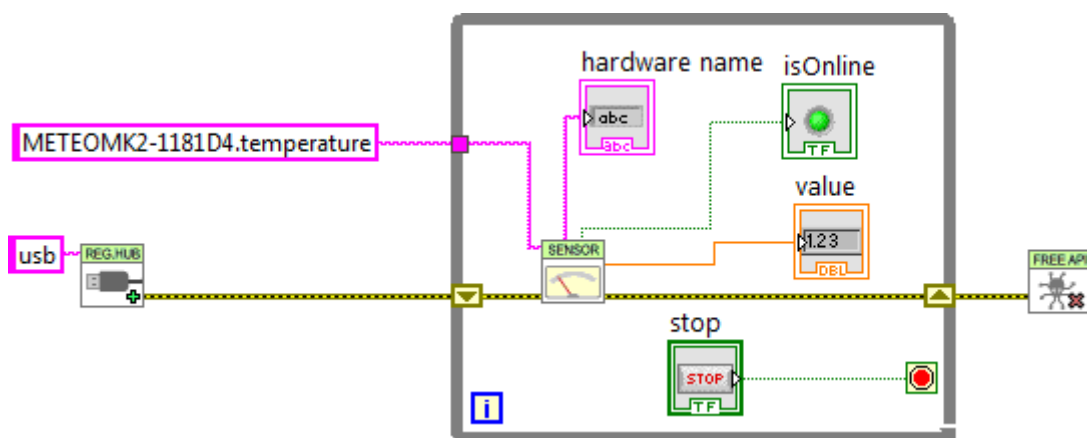
The VIs of the Yoctopuce API are actually an entry door into the library. Internally, this mechanism works independently of the Yoctopuce VIs. Indeed, most communications with electronic modules are managed automatically as background tasks. Therefore, you do not necessarily need to take any specific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



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

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

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

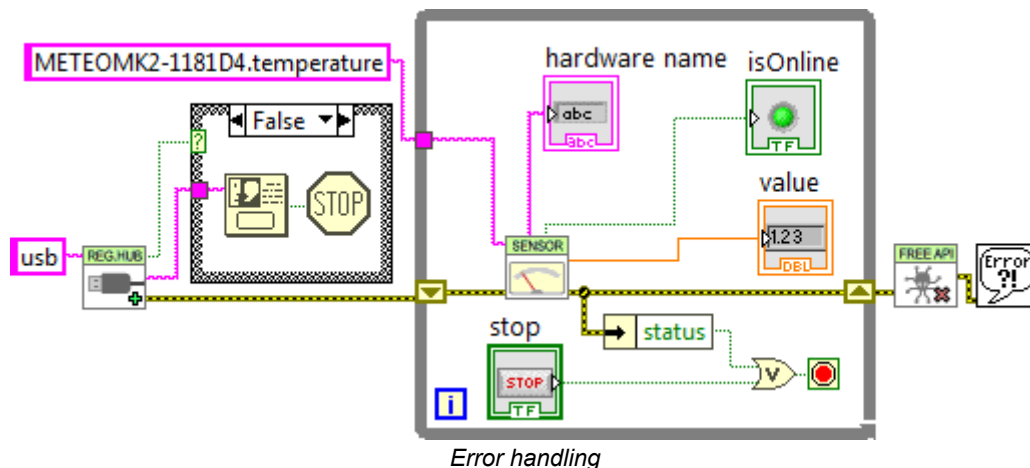


Using names to identify the functions to be used

Error handling

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

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

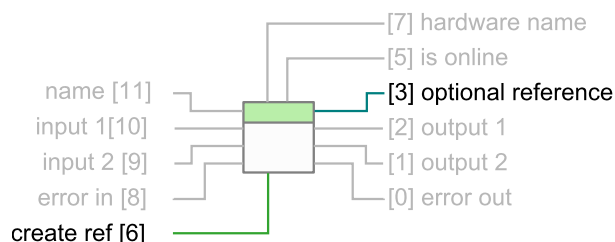


The most common error message is "Another process is already using yAPI". It means that another application, LabVIEW or other, already uses the API in native USB mode. For technical reasons, the native USB API can be used by only one application at the same time on the same machine. You can easily work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

11.6. Using Proxy objects

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

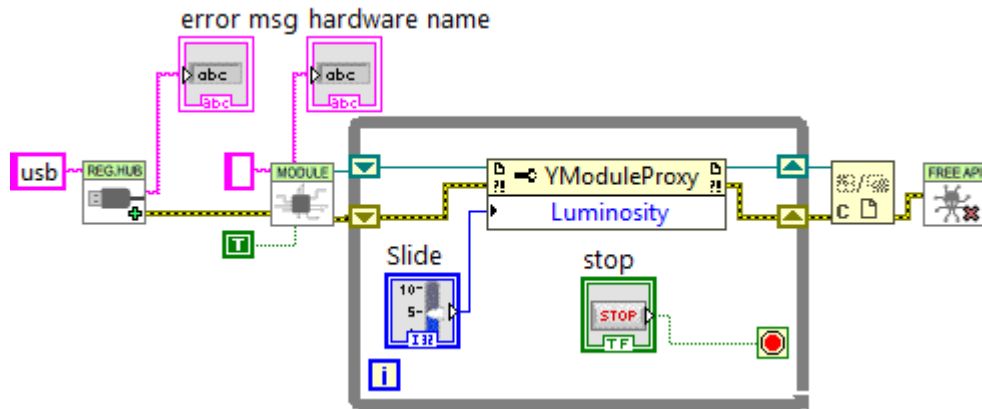
Each VI corresponding to a class has two connectors *create ref* and *optional ref* which enable you to obtain a reference on the Proxy object of the .NET Proxy API on which the LabVIEW library is built.



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

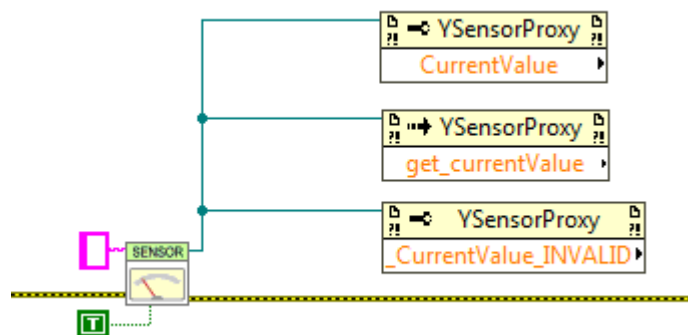
To obtain this reference, you only need to set *optional ref* to TRUE. Note, it is essential to close all references created in this way, otherwise you risk to quickly saturate the computer memory.

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



Regulating the luminosity of the leds of a module

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

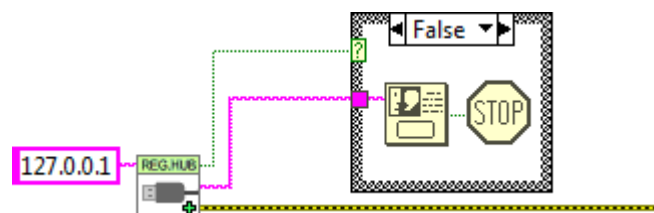


Property and Invoke nodes: Using properties, methods and constants

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

Network mode

On a given machine, there can be only one process accessing local Yoctopuce modules directly by USB (url set to "usb"). It is however possible that multiple process connect in parallel to YoctoHubs⁷ or to a machine on which *VirtualHub*⁸ is running, including the local machine. Therefore, if you use the local address of your machine (127.0.0.1) and if a VirtualHub runs on it, you can work around the limitation which prevents using the native USB API in parallel.

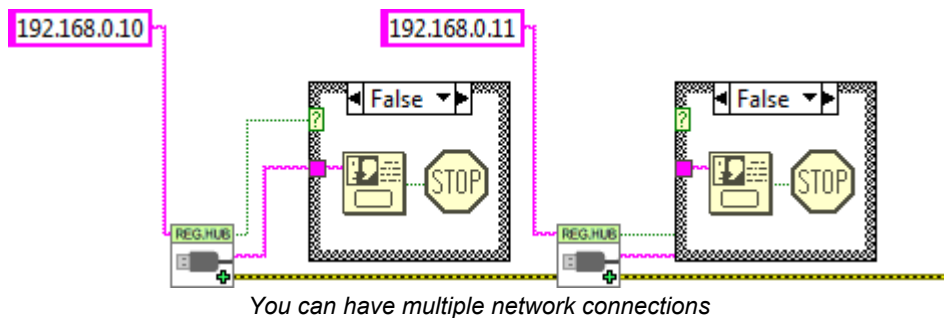


Network mode

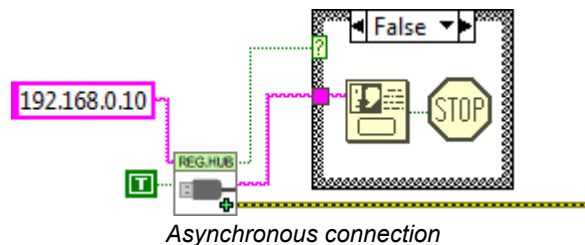
⁷ <https://www.yoctopuce.com/EN/products/category/extensions-and-networking>

⁸ www.yoctopuce.com/EN/virtualhub.php

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



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

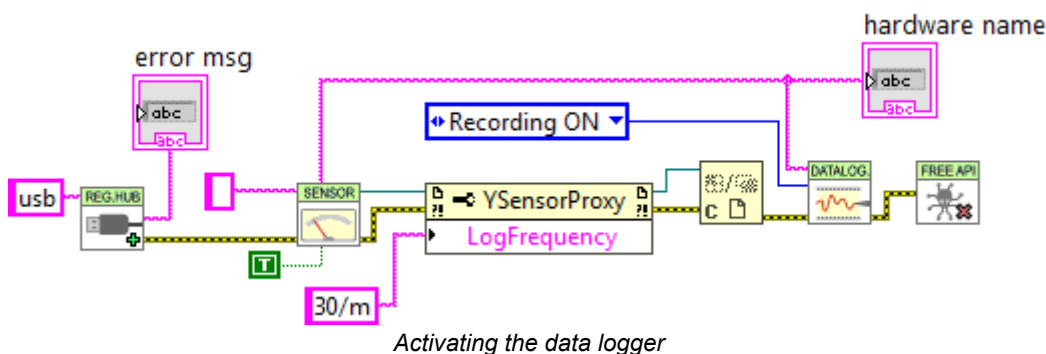


11.7. Managing the data logger

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

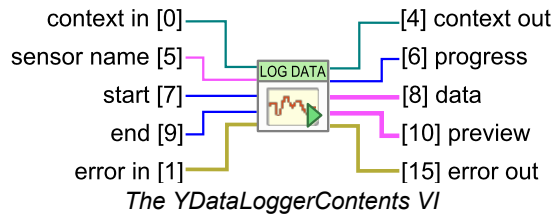
Logging

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



Reading

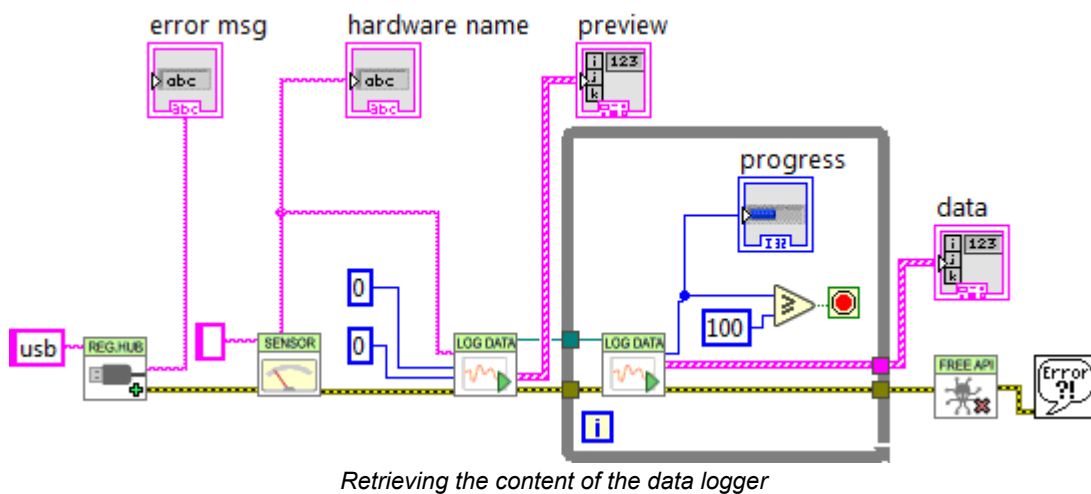
You can retrieve the data in the data logger with the YDataLoggerContents VI.



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

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

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



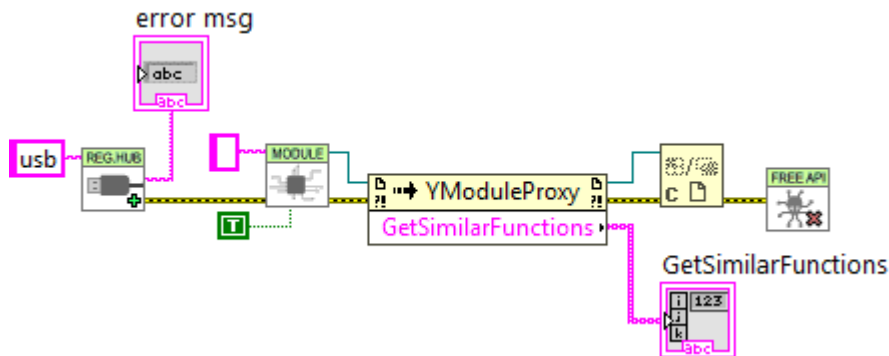
The results and the summary are returned as an array of structures containing the following fields:

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

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

11.8. Function list

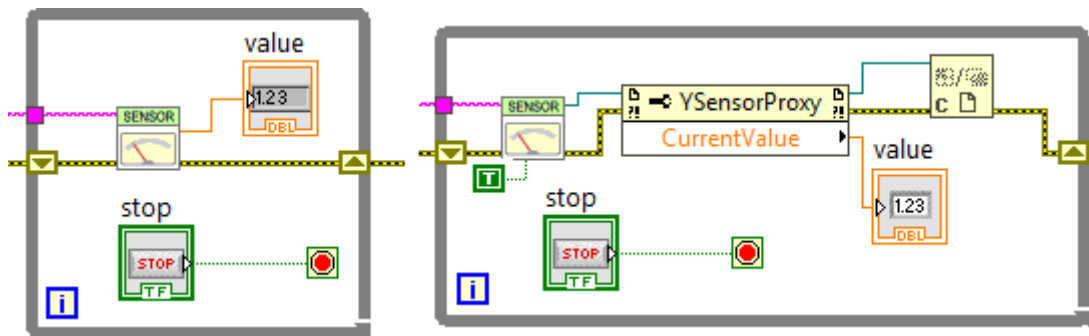
Each VI corresponding to an object of the *Proxy API* enables you to list all the functions of the same class with the *getSimilarFunctions()* method of the corresponding *Proxy* object. Thus, you can easily perform an inventory of all the connected modules, of all the connected sensors, of all the connected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



Retrieving the list of all the modules which are connected

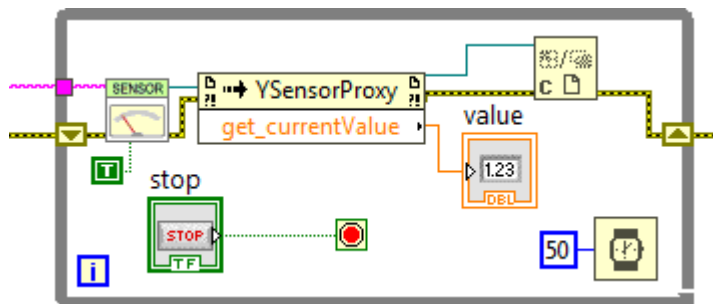
11.9. A word on performances

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



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

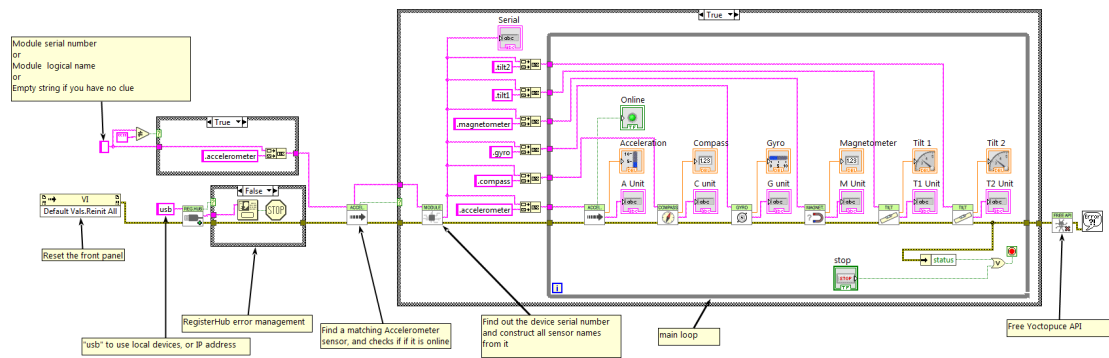
However, almost all the methods of the available Proxy objects initiate a communication with the Yoctopuce modules each time they are called. You should therefore avoid calling them too often without purpose.



This loop, using a method, must be slowed down

11.10. A full example of a LabVIEW program

Here is a short example of how to use the Yocto-3D-V2-C in LabVIEW. After a call to the *RegisterHub* VI, the *YAccelerometer* VI finds the first accelerometer sensor available, then use the *YModule* VI to find out the device serial number. This number is used to build the name of all sensors present on the device. These names are used to initialize one VI per sensor. This technique avoids ambiguities when several Yocto-3D-V2-C are connected at the same time. Once every VI is initialized, the sensor value can be displayed. When the application is about to exit, it frees the Yoctopuce API, thanks to the *YFreeAPI* VI.



Example of Yocto-3D-V2-C usage in LabVIEW

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

11.11. Differences from other Yoctopuce APIs

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

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

YFreeAPI

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

Properties

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

Callback vs. Properties

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

12. Using the Yocto-3D-V2-C with Java

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

12.1. Getting ready

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

- The Java programming library¹
- VirtualHub² for Windows, macOS or Linux, depending on your OS

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

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

12.2. Control of the Tilt function

A few lines of code are enough to use a Yocto-3D-V2-C. Here is the skeleton of a Java code snippet to use the Tilt function.

```
[...]
// Get access to your device, through the VirtualHub running locally
YAPI.RegisterHub("127.0.0.1");
[...]

// Retrieve the object used to interact with the device
tilt = YTilt.FindTilt("Y3DMK02C-123456.tilt1");

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

¹ www.yoctopuce.com/EN/libraries.php

² www.yoctopuce.com/EN/virtualhub.php

```
// Use tilt.getCurrentValue()
[...]
```

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.

YTilt.FindTilt

The `YTilt.FindTilt` function allows you to find a tilt sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-3D-V2-C module with serial number `Y3DMK02C-123456` which you have named `"MyModule"`, and for which you have given the `tilt1` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
tilt = YTilt.FindTilt("Y3DMK02C-123456.tilt1")
tilt = YTilt.FindTilt("Y3DMK02C-123456.MyFunction")
tilt = YTilt.FindTilt("MyModule.tilt1")
tilt = YTilt.FindTilt("MyModule.MyFunction")
tilt = YTilt.FindTilt("MyFunction")
```

`YTilt.FindTilt` returns an object which you can then use at will to control the tilt sensor.

isOnline

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

getCurrentValue

The `getCurrentValue()` method of the object returned by `YTilt.FindTilt` provides the value of orientation measured by the tilt sensor. The value returned is a floating number.

YCompass.FindCompass, YGyro.FindGyro...

Functions `YCompass.FindCompass`, `YMagnetometer.FindMagnetometer`, `YGyro.FindGyro` and `YAccelerometer.FindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `YTilt.FindTilt`.

A real example

Launch your Java environment and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-3D-V2-C** of the Yoctopuce library.

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

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

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);
    }

    YTilt anytilt, tilt1, tilt2;
    YCompass compass;
    YAccelerometer accelerometer;
    YGyro gyro;

    if (args.length == 0) {
        anytilt = YTilt.FirstTilt();
        if (anytilt == null) {
            System.out.println("No module connected (check USB cable)");
            System.exit(1);
        }
    } else {
        anytilt = YTilt.FindTilt(args[0] + ".tilt1");
        if (!anytilt.isOnline()) {
            System.out.println("Module not connected (check identification and USB
cable)");
            System.exit(1);
        }
    }
    try {
        String serial = anytilt.get_module().get_serialNumber();
        tilt1 = YTilt.FindTilt(serial + ".tilt1");
        tilt2 = YTilt.FindTilt(serial + ".tilt2");
        compass = YCompass.FindCompass(serial + ".compass");
        accelerometer = YAccelerometer.FindAccelerometer(serial + ".accelerometer");
        gyro = YGyro.FindGyro(serial + ".gyro");
        int count = 0;
        while (true) {
            if (!tilt1.isOnline()) {
                System.out.println("device disconnected");
                System.exit(0);
            }

            if (count % 10 == 0)
                System.out.println("tilt1    tilt2    compass    acc    gyro");

            System.out.println(" " + tilt1.get_currentValue() + "\t" +
tilt2.get_currentValue() +
"\t" + compass.get_currentValue() + "\t" +
accelerometer.get_currentValue() + "\t" +
gyro.get_currentValue());
            count++;
            YAPI.Sleep(250);
        }
    } catch (YAPI_Exception ex) {
        System.out.println("Module not connected (check identification and USB cable)"
);
    }
    YAPI.FreeAPI();
}

```

12.3. Control of the module part

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

```

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

public class Demo {

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

```

```

    } catch (YAPI_Exception ex) {
        System.out.println("Cannot contact VirtualHub on 127.0.0.1 (" +
ex.getLocalizedMessage() + ")");
        System.out.println("Ensure that the VirtualHub application is running");
        System.exit(1);
    }
    System.out.println("usage: demo [serial or logical name] [ON/OFF]");

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

    try {
        if (args.length > 1) {
            if (args[1].equalsIgnoreCase("ON")) {
                module.setBeacon(YModule.BEACON_ON);
            } else {
                module.setBeacon(YModule.BEACON_OFF);
            }
        }

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

```

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

Changing the module settings

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

```

import com.yoctopuce.YoctoAPI.*;

public class Demo {

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

```

```

    }

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

    YModule m;
    String newname;

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

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

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

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

    YAPI.FreeAPI();
}

```

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

Listing the modules

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

```

import com.yoctopuce.YoctoAPI.*;

public class Demo {

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

        System.out.println("Device list");
        YModule module = YModule.FirstModule();
        while (module != null) {
            try {
                System.out.println(module.get_serialNumber() + " (" +
module.get_productName() + ")");
            } catch (YAPI_Exception ex) {
                break;
            }
        }
    }
}

```

```
        module = module.nextModule();
    }
    YAPI.FreeAPI();
}
```

12.4. Error handling

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

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

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

13. Using the Yocto-3D-V2-C with Android

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

13.1. Native access and VirtualHub

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

13.2. Getting ready

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

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

13.3. Compatibility

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

Android version

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

¹ www.yoctopuce.com/EN/libraries.php

USB *host* support

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

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

13.4. Activating the USB port under Android

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

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

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

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

13.5. Control of the Tilt function

A few lines of code are enough to use a Yocto-3D-V2-C. Here is the skeleton of a Java code snippet to use the Tilt function.

```
[...]
// Enable detection of USB devices
YAPI.EnableUSBHost(this);
YAPI.RegisterHub("usb");
[...]
// Retrieve the object used to interact with the device
tilt = YTilt.FindTilt("Y3DMK02C-123456.tilt1");

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

[...]
```

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.

YTilt.FindTilt

The `YTilt.FindTilt` function allows you to find a tilt sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-3D-V2-C module with serial number `Y3DMK02C-123456` which you have named "MyModule", and for which you have given the *tilt1* function the name "MyFunction". The following five calls are strictly equivalent, as long as "MyFunction" is defined only once.

```
tilt = YTilt.FindTilt("Y3DMK02C-123456.tilt1")
tilt = YTilt.FindTilt("Y3DMK02C-123456.MyFunction")
tilt = YTilt.FindTilt("MyModule.tilt1")
tilt = YTilt.FindTilt("MyModule.MyFunction")
tilt = YTilt.FindTilt("MyFunction")
```

`YTilt.FindTilt` returns an object which you can then use at will to control the tilt sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `YTilt.FindTilt` provides the value of orientation measured by the tilt sensor. The value returned is a floating number.

YCompass.FindCompass, YGyro.FindGyro...

Functions `YCompass.FindCompass`, `YMagnetometer.FindMagnetometer`, `YGyro.FindGyro` and `YAccelerometer.FindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `YTilt.FindTilt`.

A real example

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

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

```
package com.yoctopuce.doc_examples;

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

import com.yoctopuce.YoctoAPI.YAPI;
import com.yoctopuce.YoctoAPI.YAPI_Exception;
import com.yoctopuce.YoctoAPI.YAccelerometer;
import com.yoctopuce.YoctoAPI.YCompass;
import com.yoctopuce.YoctoAPI.YGyro;
import com.yoctopuce.YoctoAPI.YModule;
import com.yoctopuce.YoctoAPI.YSensor;
```

```

import com.yoctopuce.YoctoAPI.YTilt;

public class GettingStarted_Yocto_3D extends Activity implements OnItemSelectedListener
{

    private ArrayAdapter<String> aa;
    private String serial = "";
    private Handler handler = null;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.gettingstarted_yocto_3d);
        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);
        handler = new Handler();
    }

    @Override
    protected void onStart()
    {
        super.onStart();
        try {
            aa.clear();
            YAPI.EnableUSBHost(this);
            YAPI.RegisterHub("usb");
            YModule module = YModule.FirstModule();
            while (module != null) {
                if (module.get_productName().equals("Yocto-3D")) {
                    String serial = module.get_serialNumber();
                    aa.add(serial);
                }
                module = module.nextModule();
            }
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
        aa.notifyDataSetChanged();
        handler.postDelayed(r, 500);
    }

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

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

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

    final Runnable r = new Runnable()
    {
        public void run()
        {
            if (serial != null) {
                YSensor tilt1 = YTilt.FindTilt(serial + ".tilt1");
                try {
                    TextView view = (TextView) findViewById(R.id.tilt1field);
                    view.setText(String.format("%.1f %s", tilt1.getCurrentValue(),
tilt1.getUnit()));
                } catch (YAPI_Exception e) {
                    e.printStackTrace();
                }
                YTilt tilt2 = YTilt.FindTilt(serial + ".tilt2");
            }
        }
    }
}

```

```

        try {
            TextView view = (TextView) findViewById(R.id.tilt2field);
            view.setText(String.format("%.1f %s", tilt2.getCurrentValue(),
tilt2.getUnit()));
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
        YCompass compass = YCompass.FindCompass(serial + ".compass");
        try {
            TextView view = (TextView) findViewById(R.id.compassfield);
            view.setText(String.format("%.1f %s", compass.getCurrentValue(),
compass.getUnit()));
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
        YAccelerometer accelerometer = YAccelerometer.FindAccelerometer(serial +
".accelerometer");
        try {
            TextView view = (TextView) findViewById(R.id.accelfield);
            view.setText(String.format("%.1f %s", accelerometer.getCurrentValue(),
accelerometer.getUnit()));
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
        YGyro gyro = YGyro.FindGyro(serial + ".gyro");
        try {
            TextView view = (TextView) findViewById(R.id.gyrofield);
            view.setText(String.format("%.1f %s", gyro.getCurrentValue(),
gyro.getUnit()));
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
    }
    handler.postDelayed(this, 200);
};
}
}

```

13.6. Control of the module part

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

```

package com.yoctopuce.doc_examples;

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

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

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

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.modulecontrol);
        Spinner my_spin = (Spinner) findViewById(R.id.spinner1);
        my_spin.setOnItemClickListener(this);
    }
}

```

```

        aa = new ArrayAdapter<String>(this, android.R.layout.simple_spinner_item);
        aa.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
        my_spin.setAdapter(aa);
    }

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

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

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

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

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

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

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

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

```

```

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

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

```

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

Changing the module settings

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

```

package com.yoctopuce.doc_examples;

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

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

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

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

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

        try {
            aa.clear();

```

```

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

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

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

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

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

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

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

```

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

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

Listing the modules

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

```
package com.yoctopuce.doc_examples;

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

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

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

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

        try {
            YAPI.UpdateDeviceList();
            YModule module = YModule.FirstModule();
            while (module != null) {
                String line = module.get_serialNumber() + " (" + module.get_productName() +
                ")";

                TextView tx = new TextView(this);
                tx.setText(line);
                tx.setTextSize(TypedValue.COMPLEX_UNIT_SP, 20);
                layout.addView(tx);
                module = module.nextModule();
            }
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
    }

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

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

```
}
```

13.7. Error handling

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

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

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

14. Using Yocto-3D-V2-C with TypeScript

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

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

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

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

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

14.1. Using the Yoctopuce library for TypeScript

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

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

```
npm install -g typescript
```

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

- The TypeScript programming library¹
- The VirtualHub software² for Windows, macOS, or Linux, depending on your OS. TypeScript and JavaScript are part of those languages which do not generally allow you to directly access to USB peripherals. Therefore the library can only be used to access network-enabled devices (connected through a YoctoHub), or USB devices accessible through Yoctopuce TCP/IP to USB gateway, named *VirtualHub*. No extra driver will be needed, though.

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

```
npm install
```

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

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

14.2. Refresher on asynchronous I/O in JavaScript

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

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

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

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

¹ www.yoctopuce.com/EN/libraries.php

² www.yoctopuce.com/EN/virtualhub.php

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

This TypeScript library uses the *Promise* objects and *async* methods, to allow you to use the *await* syntax. To keep it easy to remember, all public methods of the TypeScript library are *async*, i.e. return a Promise object, except:

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

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

14.3. Control of the Tilt function

A few lines of code are enough to use a Yocto-3D-V2-C. Here is the skeleton of a TypeScript code snippet to use the Tilt function.

```
// For Node.js, the library is referenced through the NPM package
// For HTML, we would use instead a relative path (depending on the build environment)
import { YAPI, YErrorMsg, YModule } from 'yoctolib-cjs/yocto_api_nodejs.js';
import { YTilt } from 'yoctolib-cjs/yocto_tilt.js';

[...]
```

// Get access to your device, through the VirtualHub running locally

```
await YAPI.RegisterHub('127.0.0.1');
[...]
```

// Retrieve the object used to interact with the device

```
var tilt: YTilt = YTilt.FindTilt("Y3DMK02C-123456.tilt1");
```

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

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

Let us look at these lines in more details.

yocto_api and yocto_tilt import

These two imports provide access to functions allowing you to manage Yoctopuce modules. `yocto_api` is always needed, `yocto_tilt` is necessary to manage modules containing a tilt sensor, such as Yocto-3D-V2-C. Other imports can be useful in other cases, such as `YModule` which can let you enumerate any type of Yoctopuce device.

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

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

YAPI.RegisterHub

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

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

YTilt.FindTilt

The `FindTilt` method allows you to find a tilt sensor from the serial number of the module on which it resides and from its function name. You can also use logical names, as long as you have initialized them. Let us imagine a Yocto-3D-V2-C module with serial number `Y3DMK02C-123456` which you have named `"MyModule"`, and for which you have given the `tilt1` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
tilt = YTilt.FindTilt("Y3DMK02C-123456.tilt1")
tilt = YTilt.FindTilt("Y3DMK02C-123456.MaFonction")
tilt = YTilt.FindTilt("MonModule.tilt1")
tilt = YTilt.FindTilt("MonModule.MaFonction")
tilt = YTilt.FindTilt("MaFonction")
```

`YTilt.FindTilt` returns an object which you can then use at will to control the tilt sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `YTilt.FindTilt` provides the value of orientation measured by the tilt sensor. The value returned is a floating number.

YCompass.FindCompass, YGyro.FindGyro...

Functions `YCompass.FindCompass`, `YMagnetometer.FindMagnetometer`, `YGyro.FindGyro` and `YAccelerometer.FindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `YTilt.FindTilt`.

A real example, for Node.js

Open a command window (a terminal, a shell...) and go into the directory `example_nodejs/Doc-GettingStarted-Yocto-3D-V2-C` within Yoctopuce library for TypeScript. In there, you will find a file named `demo.ts` with the sample code below, which uses the functions explained above, but this time used with all side materials needed to make it work nicely as a small demo.

If your Yocto-3D-V2-C is not connected on the host running the browser, replace in the example the address `127.0.0.1` by the IP address of the host on which the Yocto-3D-V2-C is connected and where you run the VirtualHub.

```
import { YAPI, YErrorMsg, YModule } from 'yoctolib-cjs/yocto_api_nodejs.js';
import { YTilt } from 'yoctolib-cjs/yocto_tilt.js';
import { YCompass } from 'yoctolib-cjs/yocto_compass.js';
import { YGyro } from 'yoctolib-cjs/yocto_gyro.js';
import { YAccelerometer } from 'yoctolib-cjs/yocto_accelerometer.js';

let tilt1: YTilt;
let tilt2: YTilt;
let compass: YCompass;
let gyro: YGyro;
let accelerometer: YAccelerometer;
let count: number;

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

    // Setup the API to use the VirtualHub on local machine
    let errmsg: YErrorMsg = new YErrorMsg();
```

```

if(await YAPI.RegisterHub('127.0.0.1', errmsg) !== YAPI.SUCCESS) {
    console.log('Cannot contact VirtualHub on 127.0.0.1: '+errmsg.msg);
    return;
}

// Select specified device, or use first available one
let serial: string = process.argv[process.argv.length-1];
if(serial[8] !== '-') {
    // by default use any connected module suitable for the demo
    let anysensor = YTilt.FirstTilt();
    if(anysensor) {
        let module = await anysensor.get_module();
        serial = await module.get_serialNumber();
    } else {
        console.log('No matching sensor connected, check cable !');
        await YAPI.FreeAPI();
        return;
    }
}
console.log('Using device '+serial);
tilt1 = YTilt.FindTilt(serial + ".tilt1");
tilt2 = YTilt.FindTilt(serial + ".tilt2");
compass = YCompass.FindCompass(serial + ".compass");
gyro = YGyro.FindGyro(serial + ".gyro");
accelerometer = YAccelerometer.FindAccelerometer(serial+".accelerometer");
count = 0;

refresh();
}

async function refresh(): Promise<void>
{
    if (await tilt1.isOnline()) {
        if (count % 10 == 0) {
            console.log("tilt1\ttilt2\tcompass\tacc\tgyro");
        }
        console.log(
            await tilt1.get_currentValue()+"\t"+
            await tilt2.get_currentValue()+"\t"+
            await compass.get_currentValue()+"\t"+
            await accelerometer.get_currentValue()+"\t"+
            await gyro.get_currentValue()
        );
        count++;
    } else {
        console.log('Module not connected');
        count = 0;
    }
    setTimeout(refresh, 500);
}

startDemo();

```

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

```
npm install
```

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

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

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

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

- a JavaScript file named `demo.js`, that Node.js can run

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

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

```
npm install yoctolib-cjs
```

Same example, but this time running in a browser

If you want to see how to use the library within a browser rather than with Node.js, switch to the directory **example_html/Doc-GettingStarted-Yocto-3D-V2-C**. You will find there an HTML file named `app.html`, and a TypeScript file `app.ts` similar to the code above, but with a few changes since it has to interact through an HTML page rather than through the JavaScript console.

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

```
npm run app-server
```

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

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

14.4. Control of the module part

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

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

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

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

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

```

```

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

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

```

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

Changing the module settings

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

```

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

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

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

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

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

```

Warning: the number of write cycles of the nonvolatile memory of the module is limited. When this limit is reached, nothing guaranties that the saving process is performed correctly. This limit, linked to

the technology employed by the module micro-processor, is located at about 100000 cycles. In short, you can use the `saveToFlash()` method only 100000 times in the life of the module. Make sure you do not call this method within a loop.

Listing the modules

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

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

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

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

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

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

startDemo();
```

14.5. Error handling

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

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

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

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

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

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

15. Using Yocto-3D-V2-C with JavaScript / EcmaScript

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

15.2. Using Yoctopuce library for JavaScript / EcmaScript 2017

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

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

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

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

Using the official Yoctopuce library for node.js

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

¹ www.yoctopuce.com/EN/libraries.php

² www.yoctopuce.com/EN/virtualhub.php

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

```
npm install
```

Once done, you can start the example file using:

```
node demo.js
```

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

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

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

Using the Yoctopuce library within a browser (HTML)

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

Using the Yoctoluce library on older JavaScript engines

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

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

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

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

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

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

Backward-compatibility with the old JavaScript library

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

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

by

³ <http://babeljs.io>

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

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

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

by

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

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

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

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

you can use:

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

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

You can also rewrite this last asynchronous call as:

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

15.3. Control of the Tilt function

A few lines of code are enough to use a Yocto-3D-V2-C. Here is the skeleton of a JavaScript code snippet to use the Tilt function.

```
// For Node.js, we use function require()
// For HTML, we would use <script src="...">
require('yoctolib-es2017/yocto_api.js');
require('yoctolib-es2017/yocto_tilt.js');

[...]
// Get access to your device, through the VirtualHub running locally
await YAPI.RegisterHub('127.0.0.1');
[...]

// Retrieve the object used to interact with the device
var tilt = YTilt.FindTilt("Y3DMK02C-123456.tilt1");

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

```

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

```

Let us look at these lines in more details.

yocto_api and yocto_tilt import

These two import provide access to functions allowing you to manage Yoctopuce modules. `yocto_api` is always needed, `yocto_tilt` is necessary to manage modules containing a tilt sensor, such as Yocto-3D-V2-C. Other imports can be useful in other cases, such as `YModule` which can let you enumerate any type of Yoctopuce device.

YAPI.RegisterHub

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

YTilt.FindTilt

The `FindTilt` method allows you to find a tilt sensor from the serial number of the module on which it resides and from its function name. You can also use logical names, as long as you have initialized them. Let us imagine a Yocto-3D-V2-C module with serial number `Y3DMK02C-123456` which you have named `"MyModule"`, and for which you have given the `tilt1` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```

tilt = YTilt.FindTilt("Y3DMK02C-123456.tilt1")
tilt = YTilt.FindTilt("Y3DMK02C-123456.MaFonction")
tilt = YTilt.FindTilt("MonModule.tilt1")
tilt = YTilt.FindTilt("MonModule.MaFonction")
tilt = YTilt.FindTilt("MaFonction")

```

`YTilt.FindTilt` returns an object which you can then use at will to control the tilt sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `YTilt.FindTilt` provides the value of orientation measured by the tilt sensor. The value returned is a floating number.

YCompass.FindCompass, YGyro.FindGyro...

Functions `YCompass.FindCompass`, `YMagnetometer.FindMagnetometer`, `YGyro.FindGyro` and `YAccelerometer.FindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `YTilt.FindTilt`.

A real example, for Node.js

Open a command window (a terminal, a shell...) and go into the directory **example_nodejs/Doc-GettingStarted-Yocto-3D-V2-C** within Yoctopuce library for JavaScript / EcmaScript 2017. In there, you will find a file named `demo.js` with the sample code below, which uses the functions explained above, but this time used with all side materials needed to make it work nicely as a small demo.

If your Yocto-3D-V2-C is not connected on the host running the browser, replace in the example the address 127.0.0.1 with the IP address of the host on which the Yocto-3D-V2-C is connected and where you run the VirtualHub.

```
"use strict";

require('yoctolib-es2017/yocto_api.js');
require('yoctolib-es2017/yocto_tilt.js');
require('yoctolib-es2017/yocto_compass.js');
require('yoctolib-es2017/yocto_gyro.js');
require('yoctolib-es2017/yocto_accelerometer.js');

let tilt1, tilt2, compass, gyro, accelerometer, count;

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

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

    // Select specified device, or use first available one
    let serial = process.argv[process.argv.length-1];
    if(serial[8] !== '-') {
        // by default use any connected module suitable for the demo
        let anysensor = YTilt.FirstTilt();
        if(anysensor) {
            let module = await anysensor.module();
            serial = await module.get_serialNumber();
        } else {
            console.log('No matching sensor connected, check cable !');
            return;
        }
    }
    console.log('Using device '+serial);
    tilt1 = YTilt.FindTilt(serial + ".tilt1");
    tilt2 = YTilt.FindTilt(serial + ".tilt2");
    compass = YCompass.FindCompass(serial + ".compass");
    gyro = YGyro.FindGyro(serial + ".gyro");
    accelerometer = YAccelerometer.FindAccelerometer(serial+".accelerometer");
    count = 0;

    refresh();
}

async function refresh()
{
    if (await tilt1.isOnline()) {
        if (count % 10 == 0) {
            console.log("tilt1\ttilt2\ttcompass\ttacc\ttgyro");
        }
        console.log(
            await tilt1.get_currentValue()+"\t"+
            await tilt2.get_currentValue()+"\t"+
            await compass.get_currentValue()+"\t"+
            await accelerometer.get_currentValue()+"\t"+
            await gyro.get_currentValue()
        );
        count++;
    } else {
        console.log('Module not connected');
        count = 0;
    }
    setTimeout(refresh, 500);
}

startDemo();
```

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

```
npm install
```

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

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

Same example, but this time running in a browser

If you want to see how to use the library within a browser rather than with Node.js, switch to the directory **example_html/Doc-GettingStarted-Yocto-3D-V2-C**. You will find there a single HTML file, with a JavaScript section similar to the code above, but with a few changes since it has to interact through an HTML page rather than through the JavaScript console.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Hello World</title>
  <script src="../../lib/yocto_api.js"></script>
  <script src="../../lib/yocto_tilt.js"></script>
  <script src="../../lib/yocto_compass.js"></script>
  <script src="../../lib/yocto_accelerometer.js"></script>
  <script src="../../lib/yocto_gyro.js"></script>
  <script>
    async function startDemo()
    {
      await YAPI.LogUnhandledPromiseRejections();
      await YAPI.DisableExceptions();

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

    async function refresh()
    {
      let serial = document.getElementById('serial').value;
      if(serial == '') {
        // by default use any connected module suitable for the demo
        let anysensor = YTilt.FirstTilt();
        if(anysensor) {
          let module = await anysensor.module();
          serial = await module.get_serialNumber();
          document.getElementById('serial').value = serial;
        }
      }
      let tilt1      = YTilt.FindTilt(serial+".tilt1");
      let tilt2      = YTilt.FindTilt(serial+".tilt2");
      let compass    = YCompass.FindCompass(serial+".compass");
      let gyro       = YGyro.FindGyro(serial+".gyro");
      let accelerometer = YAccelerometer.FindAccelerometer(serial+".accelerometer");
      if (await tilt1.isOnline()) {
        document.getElementById('msg').value = '';
        document.getElementById("tilt1-val").value = await tilt1.get_currentValue();
        document.getElementById("tilt2-val").value = await tilt2.get_currentValue();
        document.getElementById("compass-val").value = await compass.get_currentValue();
        document.getElementById("gyro-val").value = await gyro.get_currentValue();
        document.getElementById("accelerometer-val").value = await
        accelerometer.get_currentValue();
      } else {
        document.getElementById('msg').value = 'Module not connected';
      }
      setTimeout(refresh, 500);
    }

    startDemo();
  </script>
</head>
<body>
Module to use: <input id='serial'>
```

```
<input id='msg' style='color:red;border:none;' readonly><br>
tilt1: <input id='tilt1-val' readonly> &deg;<br>
tilt2: <input id='tilt2-val' readonly></span> &deg;<br>
compass: <input id='compass-val' readonly></span> &deg;<br>
gyro: <input id='gyro-val' readonly></span> &deg;/s<br>
accelerometer: <input id='accelerometer-val' readonly></span> g<br>
</body>
</html>
```

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

15.4. Control of the module part

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

```
"use strict";

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

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

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

    // Select the relay to use
    let module = YModule.FindModule(args[0]);
    if(await module.isOnline()) {
        if(args.length > 1) {
            if(args[1] == 'ON') {
                await module.set_beacon(YModule.BEACON_ON);
            } else {
                await module.set_beacon(YModule.BEACON_OFF);
            }
        }
        console.log('serial:      '+await module.get_serialNumber());
        console.log('logical name: '+await module.get_logicalName());
        console.log('luminosity:   '+await module.get_luminosity()+'%');
        console.log('beacon:       '+ (await module.get_beacon() == YModule.BEACON_ON
? 'ON': 'OFF'));
        console.log('upTime:       '+parseInt(await module.get_upTime()/1000)+' sec');
        console.log('USB current:  '+await module.get_usbCurrent()+' mA');
        console.log('logs:');
        console.log(await module.get_lastLogs());
    } else {
        console.log("Module not connected (check identification and USB cable)\n");
    }
    await YAPI.FreeAPI();
}

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

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

Changing the module settings

When you want to modify the settings of a module, you only need to call the corresponding `set_xxx()` function. However, this modification is performed only in the random access memory

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

```
"use strict";

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

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

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

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

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

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

Listing the modules

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

```
"use strict";

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

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

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

```

        return;
    }
    refresh();
}

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

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

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

```

15.5. Error handling

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

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

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

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

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

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

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

16. Using Yocto-3D-V2-C with PHP

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

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

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

16.1. Getting ready

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

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

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

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

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

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

² www.yoctopuce.com/EN/libraries.php

³ www.yoctopuce.com/EN/virtualhub.php

In the library archive, there are thus three subdirectories:

- php7
- php8
- phpPSR

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

16.2. Control of the Tilt function

A few lines of code are enough to use a Yocto-3D-V2-C. Here is the skeleton of a PHP code snippet to use the Tilt function.

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

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

// Retrieve the object used to interact with the device
$tilt = YTilt::FindTilt("Y3DMK02C-123456.tilt1");

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

Let's look at these lines in more details.

yocto_api.php and yocto_tilt.php

These two PHP includes provides access to the functions allowing you to manage Yoctopuce modules. `yocto_api.php` must always be included, `yocto_tilt.php` is necessary to manage modules containing a tilt sensor, such as Yocto-3D-V2-C.

YAPI::RegisterHub

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

YTilt::FindTilt

The `YTilt::FindTilt` function allows you to find a tilt sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-3D-V2-C module with serial number `Y3DMK02C-123456` which you have named `"MyModule"`, and for which you have given the `tilt1` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
$tilt = YTilt::FindTilt("Y3DMK02C-123456.tilt1");
$tilt = YTilt::FindTilt("Y3DMK02C-123456.MyFunction");
$tilt = YTilt::FindTilt("MyModule.tilt1");
$tilt = YTilt::FindTilt("MyModule.MyFunction");
$tilt = YTilt::FindTilt("MyFunction");
```

`YTilt::FindTilt` returns an object which you can then use at will to control the tilt sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `yFindTilt` provides the angle currently measured by the inclinometer. The value returned is a floating number.

yFindCompass and yFindGyro

Functions `yFindCompass`, `yFindGyro`, `yFindMagnetometer` and `yFindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `YFindTilt`.

A real example

Open your preferred text editor⁴, copy the code sample below, save it with the Yoctopuce library files in a location which is accessible to your web server, then use your preferred web browser to access this page. The code is also provided in the directory **Examples/Doc-GettingStarted-Yocto-3D-V2-C** of the Yoctopuce library.

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

```
<HTML>
<HEAD>
  <TITLE> Hello World</TITLE>
</HEAD>
<BODY>
<?php
  include('yocto_api.php');
  include('yocto_tilt.php');
  include('yocto_compass.php');
  include('yocto_gyro.php');
  include('yocto_accelerometer.php');

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

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

  @$serial = $_GET['serial'];
  if ($serial != '') {
    // Check if a specified module is available online
    $anytilt = YTilt::FindTilt("$serial.tilt1");
    if (!$anytilt->isOnline()) {
      die("Module not connected (check serial and USB cable)");
    }
  } else {
    // or use any connected module suitable for the demo
    $anytilt = YTilt::FirstTilt();
    if(is_null($anytilt)) {
      die("No module connected (check USB cable)");
    } else {
      $serial = $anytilt->module()->get_serialnumber();
    }
  }
  Print("Module to use: <input name='serial' value='$serial'><br>");

  // Get all sensor on the device matching the serial
  $tilt1 = YTilt::FindTilt("$serial.tilt1");
  $tilt2 = YTilt::FindTilt("$serial.tilt2");
  $compass = YCompass::FindCompass("$serial.compass");
  $gyro = YGyro::FindGyro("$serial.gyro");
  $accelerometer = yFindAccelerometer("$serial.accelerometer");
```

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

```

$tilt1value      = $tilt1->get_currentValue();
$tilt2value      = $tilt2->get_currentValue();
$compassvalue    = $compass->get_currentValue();
$gyrovalue       = $gyro->get_currentValue();
$accelerometervalue = $accelerometer->get_currentValue();

Print("tilt1: $tilt1value &deg;<br>");
Print("tilt2: $tilt2value &deg;<br>");
Print("compass: $compassvalue &deg;<br>");
Print("gyro: $gyrovalue &deg;/s<br>");
Print("Accelerometer: $accelerometervalue g<br>");
YAPI::FreeAPI();

// trigger auto-refresh after one second
Print("<script language='javascript1.5' type='text/JavaScript'>\n");
Print("setTimeout('window.location.reload()',500);");
Print("</script>\n");
?>
</BODY>
</HTML>

```

16.3. Control of the module part

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

```

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

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

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

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

    if (isset($_GET['beacon'])) {
      if ($_GET['beacon']=='ON')
        $module->set_beacon(Y_BEACON_ON);
      else
        $module->set_beacon(Y_BEACON_OFF);
    }
    printf('serial: %s<br>', $module->get_serialNumber());
    printf('logical name: %s<br>', $module->get_logicalName());

```

```

printf('luminosity: %s<br>', $module->get_luminosity());
print('beacon: ');
if($module->get_beacon() == Y_BEACON_ON) {
    printf("<input type='radio' name='beacon' value='ON' checked>ON ");
    printf("<input type='radio' name='beacon' value='OFF'>OFF<br>");
} else {
    printf("<input type='radio' name='beacon' value='ON'>ON ");
    printf("<input type='radio' name='beacon' value='OFF' checked>OFF<br>");
}
printf('upTime: %s sec<br>', intval($module->get_upTime()/1000));
printf('USB current: %s mA<br>', $module->get_usbCurrent());
printf('logs:<br><pre>%s</pre>', $module->get_lastLogs());
YAPI::FreeAPI();
?>
<input type='submit' value='refresh'>
</FORM>
</BODY>
</HTML>

```

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

Changing the module settings

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

```

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

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

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

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

    if (isset($_GET['newname'])) {
        $newname = $_GET['newname'];
        if (!YCheckLogicalName($newname))
            die('Invalid name');
    }

```

```

        $module->set_logicalName($newname);
        $module->saveToFlash();
    }
    printf("Current name: %s<br>", $module->get_logicalName());
    print("New name: <input name='newname' value='' maxlength=19><br>");
    YAPI::FreeAPI();
?>
<input type='submit'>
</FORM>
</BODY>
</HTML>

```

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

Listing the modules

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

```

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

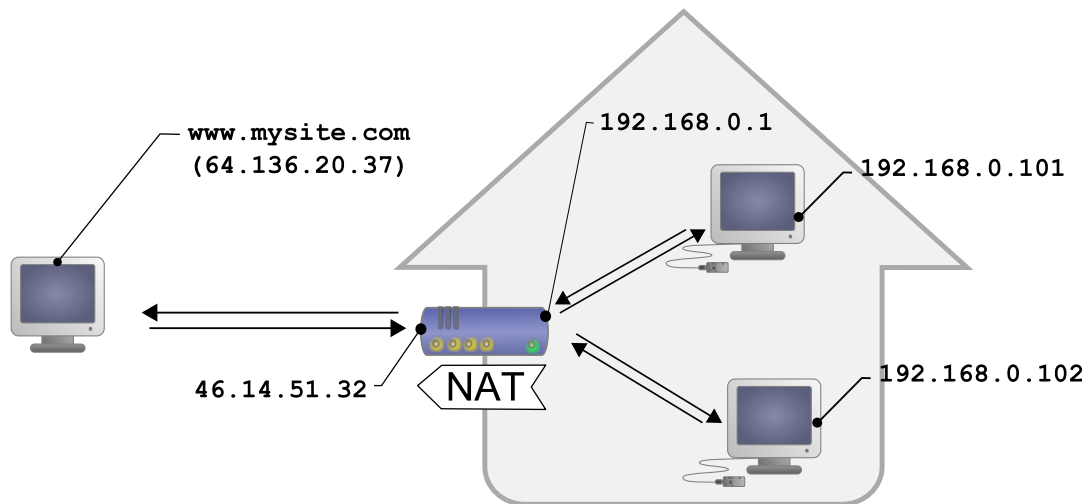
```

16.4. HTTP callback API and NAT filters

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

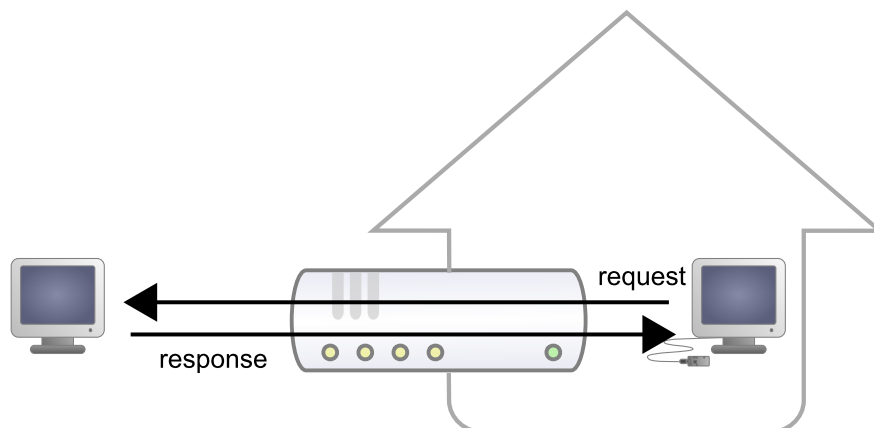
The NAT filter: advantages and disadvantages

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

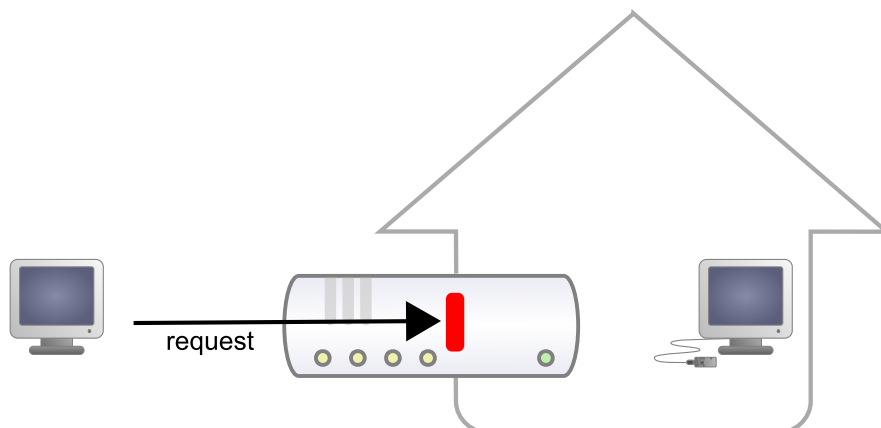


Typical DSL configuration: LAN machines are isolated from the outside by the DSL router

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



Responses from request from LAN machines are routed.

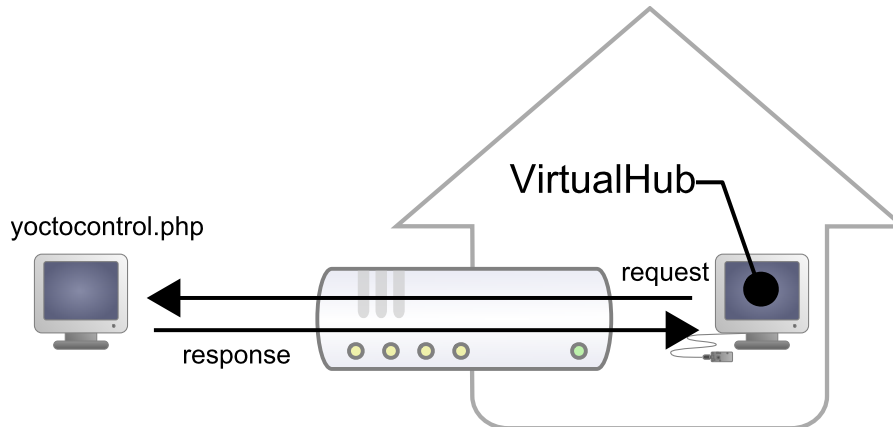


But requests from the outside are blocked.

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

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

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



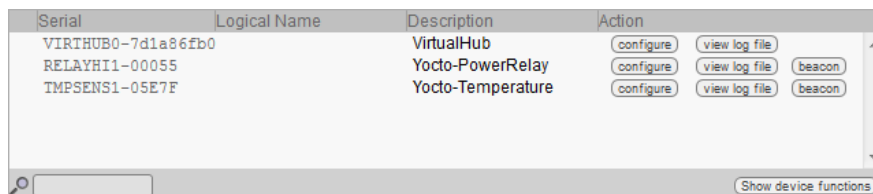
The HTTP callback API uses the *VirtualHub* which initiates the requests.

Configuration

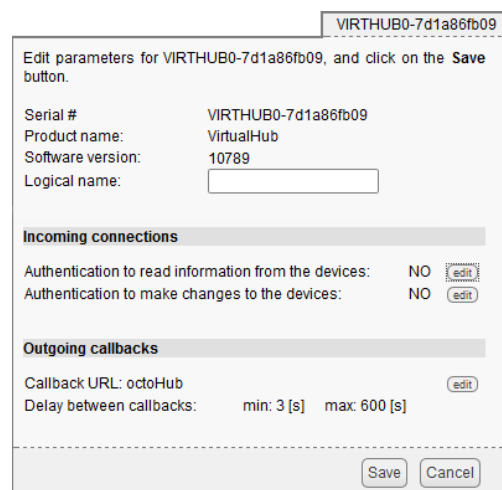
The callback API thus uses the *VirtualHub* as a gateway. All the communications are initiated by the *VirtualHub*. They are thus outgoing communications and therefore perfectly authorized by the DSL router.

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

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



Click on the "configure" button on the first line



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

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

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

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

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

Callback URL:

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

Username:
 Password:

4. Setup the desired frequency of notifications:

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

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

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

And select "Yocto-API callback".

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

Usage

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

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

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

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

Common issues

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

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

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

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

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

Limitations

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

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

16.5. Error handling

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

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

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

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

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

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

17. Using Yocto-3D-V2-C with Visual Basic .NET

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

17.1. Installation

Download the Visual Basic Yoctopuce library from the Yoctopuce web site¹. There is no setup program, simply copy the content of the zip file into the directory of your choice. You mostly need the content of the `Sources` directory. The other directories contain the documentation and a few sample programs. All sample projects are Visual Basic 2010, projects, if you are using a previous version, you may have to recreate the projects structure from scratch.

17.2. Using the Yoctopuce API in a Visual Basic project

The Visual Basic.NET Yoctopuce library is composed of a DLL and of source files in Visual Basic. The DLL is not a .NET DLL, but a classic DLL, written in C, which manages the low level communications with the modules². The source files in Visual Basic manage the high level part of the API. Therefore, you need both this DLL and the .vb files of the `sources` directory to create a project managing Yoctopuce modules.

Configuring a Visual Basic project

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

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

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

¹ www.yoctopuce.com/EN/libraries.php

² The sources of this DLL are available in the C++ API

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

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

17.3. Control of the Tilt function

A few lines of code are enough to use a Yocto-3D-V2-C. Here is the skeleton of a Visual Basic code snippet to use the Tilt function.

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

' Retrieve the object used to interact with the device
Dim tilt As YTilt
tilt = YTilt.FindTilt("Y3DMK02C-123456.tilt1")

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

[...]
```

Let's look at these lines in more details.

YAPI.RegisterHub

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

YTilt.FindTilt

The `YTilt.FindTilt` function allows you to find a tilt sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-3D-V2-C module with serial number `Y3DMK02C-123456` which you have named `"MyModule"`, and for which you have given the `tilt1` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
tilt = YTilt.FindTilt("Y3DMK02C-123456.tilt1")
tilt = YTilt.FindTilt("Y3DMK02C-123456.MyFunction")
tilt = YTilt.FindTilt("MyModule.tilt1")
tilt = YTilt.FindTilt("MyModule.MyFunction")
tilt = YTilt.FindTilt("MyFunction")
```

`YTilt.FindTilt` returns an object which you can then use at will to control the tilt sensor.

isOnline

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

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

get_currentValue

The `get_currentValue()` method of the object returned by `yFindTilt` provides the angle currently measured by the inclinometer. The value returned is a floating number.

yFindCompass and yFindGyro

Functions `yFindCompass`, `yFindGyro`, `yFindMagnetometer` and `yFindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `YFindTilt`.

A real example

Launch Microsoft VisualBasic and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-3D-V2-C** of the Yoctopuce library.

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

```
Module Module1

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

    Sub Main()
        Dim argv() As String = System.Environment.GetCommandLineArgs()
        Dim errmsg As String = ""
        Dim target As String
        Dim serial As String
        Dim count As Integer
        Dim anytilt, tilt1, tilt2 As YTilt
        Dim compass As YCompass
        Dim accelerometer As YAccelerometer
        Dim gyro As YGyro

        If argv.Length < 2 Then Usage()

        target = argv(1)

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

        If target = "any" Then
            anytilt = YTilt.FirstTilt()
            If anytilt Is Nothing Then
                Console.WriteLine("No module connected (check USB cable)")
            End If
        Else
            anytilt = YTilt.FindTilt(target + ".tilt1")
            If Not (anytilt.IsOnline()) Then
                Console.WriteLine("Module not connected (check identification and USB cable)")
            End If
        End If

        serial = anytilt.get_module().get_serialNumber()
        tilt1 = YTilt.FindTilt(serial + ".tilt1")
        tilt2 = YTilt.FindTilt(serial + ".tilt2")
        compass = YCompass.FindCompass(serial + ".compass")
        accelerometer = YAccelerometer.FindAccelerometer(serial + ".accelerometer")
        gyro = YGyro.FindGyro(serial + ".gyro")
        count = 0
    End Sub

End Module
```

```

While (True)
    If (Not tilt1.isOnline()) Then
        Console.WriteLine("Module disconnected")
    End
End If

If (count Mod 10 = 0) Then
    Console.WriteLine("tilt1" + Chr(9) + "tilt2" + Chr(9) + "compass" _
        + Chr(9) + "acc" + Chr(9) + "gyro")

End If
Console.Write(tilt1.get_currentValue().ToString() + Chr(9))
Console.Write(tilt2.get_currentValue().ToString() + Chr(9))
Console.Write(compass.get_currentValue().ToString() + Chr(9))
Console.Write(accelerometer.get_currentValue().ToString() + Chr(9))
Console.WriteLine(gyro.get_currentValue().ToString())
count = count + 1
YAPI.Sleep(250, errmsg)
End While
YAPI.FreeAPI()
End Sub

End Module

```

17.4. Control of the module part

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

```

Imports System.IO
Imports System.Environment

Module Module1

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

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

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

If argv.Length < 2 Then usage()

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

```

```

End If
YAPI.FreeAPI()
End Sub

End Module

```

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

Changing the module settings

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

```

Module Module1

Sub usage()

    Console.WriteLine("usage: demo <serial or logical name> <new logical name>")
End
End Sub

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

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

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

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

End Sub

End Module

```

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

Listing the modules

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

```
Module Module1

    Sub Main()
        Dim M As ymodule
        Dim errmsg As String = ""

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

        Console.WriteLine("Device list")
        M = YModule.FirstModule()
        While M IsNot Nothing
            Console.WriteLine(M.get_serialNumber() + " (" + M.get_productName() + ")")
            M = M.nextModule()
        End While
        YAPI.FreeAPI()
    End Sub

End Module
```

17.5. Error handling

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

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

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

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

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

risk crashing your program. At worst, if you display the value without testing it, it will be outside the expected bounds for the returned value. In the case of functions which do not normally return information, the return value is `YAPI_SUCCESS` if everything went well, and a different error code in case of failure.

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

18. Using Yocto-3D-V2-C with Delphi or Lazarus

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

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

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

18.1. Preparation

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

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

Windows

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

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

² www.lazarus-ide.org

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

⁴ www.yoctopuce.com/EN/libraries.php

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

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

Linux

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

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

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

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

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

18.2. About examples

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

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

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

18.3. Control of the Tilt function

A few lines of code are enough to use a Yocto-3D-V2-C. Here is the skeleton of a Delphi code snippet to use the Tilt function.

```
uses yocto_api, yocto_tilt;

var errmsg: string;
    tilt: TYTilt;

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

// Retrieve the object used to interact with the device
tilt = yFindTilt("Y3DMK02C-123456.tilt1")

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

Let's look at these lines in more details.

yocto_api and yocto_tilt

These two units provide access to the functions allowing you to manage Yoctopuce modules. *yocto_api* must always be used, *yocto_tilt* is necessary to manage modules containing a tilt sensor, such as Yocto-3D-V2-C.

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

yRegisterHub

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

yFindTilt

The `yFindTilt` function allows you to find a tilt sensor from the serial number of the module on which it resides and from its function name. You can also use logical names, as long as you have initialized them. Let us imagine a Yocto-3D-V2-C module with serial number `Y3DMK02C-123456` which you have named `"MyModule"`, and for which you have given the `tilt1` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
tilt := yFindTilt("Y3DMK02C-123456.tilt1");
tilt := yFindTilt("Y3DMK02C-123456.MyFunction");
tilt := yFindTilt("MyModule.tilt1");
tilt := yFindTilt("MyModule.MyFunction");
tilt := yFindTilt("MyFunction");
```

`yFindTilt` returns an object which you can then use at will to control the tilt sensor.

isOnline

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

get_currentValue

The `get_currentValue()` method of the object returned by `yFindTilt` provides the angle currently measured by the inclinometer. The value returned is a floating number.

yFindCompass and yFindGyro

Functions `yFindCompass`, `yFindGyro`, `yFindMagnetometer` and `yFindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `yFindTilt`.

A real example

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

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

```
program helloworld;
{$APPTYPE CONSOLE}
uses
  SysUtils,
  {$IFDEF UNIX}
  windows,
  {$ENDIF UNIX}
  yocto_api,
  yocto_tilt,
  yocto_compass,
  yocto_accelerometer,
  yocto_gyro;

Procedure Usage();
var
  exe : string;
begin
  exe:= ExtractFileName(paramstr(0));
  WriteLn(exe+' <serial_number>');
  WriteLn(exe+' <logical_name>');
```

```

    WriteLn(exe+' any');
    sleep(3000);
    halt;
End;

var
    m                : TYmodule;
    anytilt,tilt1,tilt2 : TYTilt;
    Compass          : TYcompass;
    accelerometer     : TYAccelerometer;
    gyro              : TYGyro;
    errmsg,serial     : string;
    done              : boolean;
    count             : 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);
        sleep(3000);
        exit;
    end;

    if paramstr(1)='any' then
    begin
        // lets try to find the first available tilt sensor
        anytilt := yFirstTilt();
        if anytilt=nil then
        begin
            writeln('No module connected (check USB cable)');
            sleep(3000);
            halt;
        end
    end
    else
        // or the one specified on command line
        anytilt:= YFindTilt(paramstr(1)+'.tilt1');

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

    // lets find the parent module so we can get the other sensors
    m := anytilt.get_module();
    serial := m.get_serialNumber();

    // retrieve some sensors present on the yocto-3D
    tilt1 := yFindTilt(serial+'.tilt1');
    tilt2 := yFindTilt(serial+'.tilt2');
    compass := yFindCompass(serial+'.compass');
    accelerometer := yFindaccelerometer(serial+'.accelerometer');
    gyro :=yFindGyro(serial+'.gyro');

    // let's poll
    done := false;
    count :=0;

    repeat
        if (tilt1.isOnline()) then
        begin
            if (count mod 10=0) then WriteLn('tilt1'#9'tilt2'#9'compass'#9'acc'#9'gyro');
            Write(FloatToStr(tilt1.get_currentValue())+#9);
            Write(FloatToStr(tilt2.get_currentValue())+#9);
            Write(FloatToStr(compass.get_currentValue())+#9);
            Write(FloatToStr(accelerometer.get_currentValue())+#9);
            WriteLn(FloatToStr(gyro.get_currentValue()));
            inc(count);
            Sleep(100);
        end
    else
        begin

```

```

        Writeln('Module not connected (check identification and USB cable)');
    done := true;
end;
until done;
yFreeAPI();
end.

```

18.4. Control of the module part

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

```

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

const
    serial = 'Y3DMK02C-123456'; // use serial number or logical name

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

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

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

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

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

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

```

```
yFreeAPI();
end.
```

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

Changing the module settings

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

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

const
  serial = 'Y3DMK02C-123456'; // use serial number or logical name

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

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

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

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

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

Listing the modules

Obtaining the list of the connected modules is performed with the `yFirstModule()` function which returns the first module found. Then, you only need to call the `nextModule()` function of this

object to find the following modules, and this as long as the returned value is not `nil`. Below a short example listing the connected modules.

```

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

var
  module : TYModule;
  errmsg : string;

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

  Writeln('Device list');

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

```

18.5. Error handling

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

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

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

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

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

so on. In any case, the returned value is of the expected type and is not a null pointer which would risk crashing your program. At worst, if you display the value without testing it, it will be outside the expected bounds for the returned value. In the case of functions which do not normally return information, the return value is `YAPI_SUCCESS` if everything went well, and a different error code in case of failure.

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

19. Using the Yocto-3D-V2-C with Universal Windows Platform

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

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

19.1. Blocking and asynchronous functions

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

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

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

Example:

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

    return result;
}

int res = await MyFunction(1234);
```

¹ <https://www.visualstudio.com/vs/cordova/vs/>

Our library follows these two rules and can therefore use the `await` notation.

For you not to have to wonder whether a function is asynchronous or not, there is the following convention: **all the public methods** of the UWP library **are asynchronous**, that is that you must call them with the `await` keyword, **except**:

- `GetTickCount()`, because measuring time in an asynchronous manner does not make a lot of sense...
- `FindModule()`, `FirstModule()`, `nextModule()`,... because detecting and enumerating modules is performed as a background task on internal structures which are managed transparently. It is therefore not necessary to use blocking functions while going through the lists of modules.

19.2. Installation

Download the Yoctopuce library for Universal Windows Platform from the Yoctopuce web site². There is no installation software, simply copy the content of the zip file in a directory of your choice. You essentially need the content of the `Sources` directory. The other directories contain documentation and a few sample programs. Sample projects are Visual Studio 2017 projects. Visual Studio 2017 is available on the Microsoft web site³.

19.3. Using the Yoctopuce API in a Visual Studio project

Start by creating your project. Then, from the **Solution Explorer** panel right click on your project and select **Add** then **Existing element**.

A file chooser opens: select all the files in the library `Sources` directory.

You then have the choice between simply adding the files to your project or adding them as a link (the **Add** button is actually a drop-down menu). In the first case, Visual Studio copies the selected files into your project. In the second case, Visual Studio simply creates a link to the original files. We recommend to use links, as a potential library update is thus much easier.

The Package.appxmanifest file

By default a Universal Windows application doesn't have access rights to the USB ports. If you want to access USB devices, you must imperatively declare it in the `Package.appxmanifest` file.

Unfortunately, the edition window of this file doesn't allow this operation and you must modify the `Package.appxmanifest` file by hand. In the "Solution Explorer" panel, right click on the `Package.appxmanifest` and select "View Code".

In this XML file, we must add a `DeviceCapability` node in the `Capabilities` node. This node must have a "Name" attribute with a "humaninterfacedevice" value.

Inside this node, you must declare all the modules that can be used. Concretely, for each module, you must add a "Device" node with an "Id" attribute, which has for value a character string "vidpid:USB_VENDORID USB_DEVICE_ID". The Yoctopuce USB_VENDORID is 24e0 and you can find the USB_DEVICE_ID of each Yoctopuce device in the documentation in the "Characteristics" section. Finally, the "Device" node must contain a "Function" node with the "Type" attribute with a value of "usage:ff00 0001".

For the Yocto-3D-V2-C, here is what you must add in the "Capabilities" node:

```
<DeviceCapability Name="humaninterfacedevice">
  <!-- Yocto-3D-V2-C -->
  <Device Id="vidpid:24e0 00D0">
    <Function Type="usage:ff00 0001" />
  </Device>
</DeviceCapability>
```

² www.yoctopuce.com/EN/libraries.php

³ <https://www.visualstudio.com/downloads/>

```
</Device>
</DeviceCapability>
```

Unfortunately, it's not possible to write a rule authorizing all Yoctopuce modules. Therefore, you must imperatively add each module that you want to use.

19.4. Control of the Tilt function

A few lines of code are enough to use a Yocto-3D-V2-C. Here is the skeleton of a C# code snippet to use the Tilt function.

```
[...]
// Enable detection of USB devices
await YAPI.RegisterHub("usb");
[...]

// Retrieve the object used to interact with the device
YTilt tilt = YTilt.FindTilt("Y3DMK02C-123456.tilt1");

// Hot-plug is easy: just check that the device is online
if (await tilt.isOnline())
{
    // Use tilt.get_currentValue()
    [...]
}

[...]
```

Let us look at these lines in more details.

YAPI.RegisterHub

The `YAPI.RegisterHub` function initializes the Yoctopuce API and indicates where the modules should be looked for. The parameter is the address of the virtual hub able to see the devices. If the string "usb" is passed as parameter, the API works with modules locally connected to the machine. If the initialization does not succeed, an exception is thrown.

YTilt.FindTilt

The `YTilt.FindTilt` function allows you to find a tilt sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-3D-V2-C module with serial number `Y3DMK02C-123456` which you have named "MyModule", and for which you have given the `tilt1` function the name "MyFunction". The following five calls are strictly equivalent, as long as "MyFunction" is defined only once.

```
tilt = YTilt.FindTilt("Y3DMK02C-123456.tilt1");
tilt = YTilt.FindTilt("Y3DMK02C-123456.MaFonction");
tilt = YTilt.FindTilt("MonModule.tilt1");
tilt = YTilt.FindTilt("MonModule.MaFonction");
tilt = YTilt.FindTilt("MaFonction");
```

`YTilt.FindTilt` returns an object which you can then use at will to control the tilt sensor.

isOnline

The `isOnline()` method of the object returned by `YTilt.FindTilt` allows you to know if the corresponding module is present and in working order.

get_currentValue

The `get_currentValue()` method of the object returned by `YTilt.FindTilt` provides the value of orientation measured by the tilt sensor. The value returned is a floating number.

YCompass.FindCompass, YGyro.FindGyro...

Functions `YCompass.FindCompass`, `YMagnetometer.FindMagnetometer`, `YGyro.FindGyro` and `YAccelerometer.FindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `YTilt.FindTilt`.

19.5. A real example

Launch Visual Studio and open the corresponding project provided in the directory **Examples/Doc-GettingStarted-Yocto-3D-V2-C** of the Yoctopuce library.

Visual Studio projects contain numerous files, and most of them are not linked to the use of the Yoctopuce library. To simplify reading the code, we regrouped all the code that uses the library in the `Demo` class, located in the `demo.cs` file. Properties of this class correspond to the different fields displayed on the screen, and the `Run()` method contains the code which is run when the "Start" button is pushed.

In this example, you can recognize the functions explained above, but this time used with all the side materials needed to make it work nicely as a small demo.

```
using System;
using System.Diagnostics;
using System.Threading.Tasks;
using Windows.UI.Xaml.Controls;
using com.yoctopuce.YoctoAPI;

namespace Demo
{
    public class Demo : DemoBase
    {
        public string HubURL { get; set; }
        public string Target { get; set; }

        public override async Task<int> Run()
        {
            try {
                await YAPI.RegisterHub(HubURL);

                YTilt anytilt, tilt1, tilt2;
                YCompass compass;
                YAccelerometer accel;
                YGyro gyro;

                if (Target.ToLower() == "any") {
                    anytilt = YTilt.FirstTilt();
                    if (anytilt == null) {
                        WriteLine("No module connected (check USB cable)");
                        return -1;
                    }
                } else {
                    anytilt = YTilt.FindTilt(Target + ".tilt1");
                }

                string serial = await (await anytilt.get_module()).get_serialNumber();
                tilt1 = YTilt.FindTilt(serial + ".tilt1");
                tilt2 = YTilt.FindTilt(serial + ".tilt2");
                compass = YCompass.FindCompass(serial + ".compass");
                accel = YAccelerometer.FindAccelerometer(serial + ".accelerometer");
                gyro = YGyro.FindGyro(serial + ".gyro");
                int count = 0;

                while (await tilt1.isOnline()) {
                    if (count++ % 10 == 0) {
                        WriteLine("tilt1    tilt2    compass    acc    gyro");
                    }
                    Write(await tilt1.get_currentValue() + "\t");
                    Write(await tilt2.get_currentValue() + "\t");
                    Write(await compass.get_currentValue() + "\t");
                    Write(await accel.get_currentValue() + "\t");
                    WriteLine("" + await gyro.get_currentValue());
                    await YAPI.Sleep(250);
                }
            }
        }
    }
}
```

```

    }

    WriteLine("Module not connected (check identification and USB cable)");
} catch (YAPI_Exception ex) {
    WriteLine("error: " + ex.Message);
}

YAPI.FreeAPI();
return 0;
}
}
}

```

19.6. Control of the module part

Each module can be controlled in a similar manner, you can find below a simple sample program displaying the main parameters of the module and enabling you to activate the localization beacon.

```

using System;
using System.Diagnostics;
using System.Threading.Tasks;
using Windows.UI.Xaml.Controls;
using com.yoctopuce.YoctoAPI;

namespace Demo
{
    public class Demo : DemoBase
    {
        public string HubURL { get; set; }
        public string Target { get; set; }
        public bool Beacon { get; set; }

        public override async Task<int> Run()
        {
            YModule m;
            string errormsg = "";

            if (await YAPI.RegisterHub(HubURL) != YAPI.SUCCESS) {
                WriteLine("RegisterHub error: " + errormsg);
                return -1;
            }
            m = YModule.FindModule(Target + ".module"); // use serial or logical name
            if (await m.isOnline()) {
                if (Beacon) {
                    await m.set_beacon(YModule.BEACON_ON);
                } else {
                    await m.set_beacon(YModule.BEACON_OFF);
                }

                WriteLine("serial: " + await m.get_serialNumber());
                WriteLine("logical name: " + await m.get_logicalName());
                WriteLine("luminosity: " + await m.get_luminosity());
                Write("beacon: ");
                if (await m.get_beacon() == YModule.BEACON_ON)
                    WriteLine("ON");
                else
                    WriteLine("OFF");
                WriteLine("upTime: " + (await m.get_upTime() / 1000) + " sec");
                WriteLine("USB current: " + await m.get_usbCurrent() + " mA");
                WriteLine("Logs:\r\n" + await m.get_lastLogs());
            } else {
                WriteLine(Target + " not connected on" + HubURL +
                    "(check identification and USB cable)");
            }
            await YAPI.FreeAPI();
            return 0;
        }
    }
}

```

Each property `xxx` of the module can be read thanks to a method of type `YModule.get_xxx()`, and properties which are not read-only can be modified with the help of the `YModule.set_xxx()` method. For more details regarding the used functions, refer to the API chapters.

Changing the module settings

When you want to modify the settings of a module, you only need to call the corresponding `YModule.set_XXX()` function. However, this modification is performed only in the random access memory (RAM) of the module: if the module is restarted, the modifications are lost. To memorize them persistently, it is necessary to ask the module to save its current configuration in its permanent memory. To do so, use the `YModule.saveToFlash()` method. Inversely, it is possible to force the module to forget its current settings by using the `YModule.revertFromFlash()` method. The short example below allows you to modify the logical name of a module.

```
using System;
using System.Diagnostics;
using System.Threading.Tasks;
using Windows.UI.Xaml.Controls;
using com.yoctopuce.YoctoAPI;

namespace Demo
{
    public class Demo : DemoBase
    {
        public string HubURL { get; set; }
        public string Target { get; set; }
        public string LogicalName { get; set; }

        public override async Task<int> Run()
        {
            try {
                YModule m;

                await YAPI.RegisterHub(HubURL);

                m = YModule.FindModule(Target); // use serial or logical name
                if (await m.isOnline()) {
                    if (!YAPI.CheckLogicalName(LogicalName)) {
                        WriteLine("Invalid name (" + LogicalName + ")");
                        return -1;
                    }

                    await m.set_logicalName(LogicalName);
                    await m.saveToFlash(); // do not forget this
                    Write("Module: serial= " + await m.get_serialNumber());
                    WriteLine(" / name= " + await m.get_logicalName());
                } else {
                    Write("not connected (check identification and USB cable)");
                }
            } catch (YAPI.Exception ex) {
                WriteLine("RegisterHub error: " + ex.Message);
            }
            await YAPI.FreeAPI();
            return 0;
        }
    }
}
```

Warning: the number of write cycles of the nonvolatile memory of the module is limited. When this limit is reached, nothing guaranties that the saving process is performed correctly. This limit, linked to the technology employed by the module micro-processor, is located at about 100000 cycles. In short, you can use the `YModule.saveToFlash()` function only 100000 times in the life of the module. Make sure you do not call this function within a loop.

Listing the modules

Obtaining the list of the connected modules is performed with the `YModule.yFirstModule()` function which returns the first module found. Then, you only need to call the `nextModule()` function of this object to find the following modules, and this as long as the returned value is not null. Below a short example listing the connected modules.

```
using System;
using System.Diagnostics;
using System.Threading.Tasks;
using Windows.UI.Xaml.Controls;
using com.yoctopuce.YoctoAPI;
```

```

namespace Demo
{
    public class Demo : DemoBase
    {
        public string HubURL { get; set; }

        public override async Task<int> Run()
        {
            YModule m;
            try {
                await YAPI.RegisterHub(HubURL);

                WriteLine("Device list");
                m = YModule.FirstModule();
                while (m != null) {
                    WriteLine(await m.get_serialNumber()
                        + " (" + await m.get_productName() + ")");
                    m = m.nextModule();
                }
            } catch (YAPI_Exception ex) {
                WriteLine("Error:" + ex.Message);
            }
            await YAPI.FreeAPI();
            return 0;
        }
    }
}

```

19.7. Error handling

When you implement a program which must interact with USB modules, you cannot disregard error handling. Inevitably, there will be a time when a user will have unplugged the device, either before running the software, or even while the software is running. The Yoctopuce library is designed to help you support this kind of behavior, but your code must nevertheless be conceived to interpret in the best possible way the errors indicated by the library.

The simplest way to work around the problem is the one used in the short examples provided in this chapter: before accessing a module, check that it is online with the `isOnline` function, and then hope that it will stay so during the fraction of a second necessary for the following code lines to run. This method is not perfect, but it can be sufficient in some cases. You must however be aware that you cannot completely exclude an error which would occur after the call to `isOnline` and which could crash the software.

In the Universal Windows Platform library, error handling is implemented with exceptions. You must therefore intercept and correctly handle these exceptions if you want to have a reliable project which does not crash as soon as you disconnect a module.

Library thrown exceptions are always of the `YAPI_Exception` type, so you can easily separate them from other exceptions in a `try{...} catch{...}` block.

Example:

```

try {
    ....
} catch (YAPI_Exception ex) {
    Debug.WriteLine("Exception from Yoctopuce lib:" + ex.Message);
} catch (Exception ex) {
    Debug.WriteLine("Other exceptions :" + ex.Message);
}

```


20. Using Yocto-3D-V2-C with Objective-C

Objective-C is language of choice for programming on macOS, due to its integration with the Cocoa framework. Yoctopuce supports the XCode versions supported by Apple. The Yoctopuce library is ARC compatible. You can therefore implement your projects either using the traditional *retain / release* method, or using the *Automatic Reference Counting*.

Yoctopuce Objective-C libraries¹ are integrally provided as source files. A section of the low-level library is written in pure C, but you should not need to interact directly with it: everything was done to ensure the simplest possible interaction from Objective-C.

You will soon notice that the Objective-C API defines many functions which return objects. You do not need to deallocate these objects yourself, the API does it automatically at the end of the application.

In order to keep them simple, all the examples provided in this documentation are console applications. Naturally, the libraries function in a strictly identical manner if you integrate them in an application with a graphical interface. You can find on Yoctopuce blog a detailed example² with video shots showing how to integrate the library into your projects.

20.1. Control of the Tilt function

A few lines of code are enough to use a Yocto-3D-V2-C. Here is the skeleton of a Objective-C code snippet to use the Tilt function.

```
#import "yocto_api.h"
#import "yocto_tilt.h"

...
NSError *error;
[YAPI RegisterHub:@"usb": &error]
...
// On récupère l'objet représentant le module (ici connecté en local sur USB)
tilt = [YTilt FindTilt:@"Y3DMK02C-123456.tilt1"];

// Pour gérer le hot-plug, on vérifie que le module est là
if([tilt isOnline])
{
    // Utiliser [tilt get_currentValue]
    ...
}
```

¹ www.yoctopuce.com/EN/libraries.php

² www.yoctopuce.com/EN/article/new-objective-c-library-for-mac-os-x

Let's look at these lines in more details.

yocto_api.h and yocto_tilt.h

These two import files provide access to the functions allowing you to manage Yoctopuce modules. `yocto_api.h` must always be used, `yocto_tilt.h` is necessary to manage modules containing a tilt sensor, such as Yocto-3D-V2-C.

[YAPI RegisterHub]

The `[YAPI RegisterHub]` function initializes the Yoctopuce API and indicates where the modules should be looked for. When used with the parameter `@"usb"`, it will use the modules locally connected to the computer running the library. If the initialization does not succeed, this function returns a value different from `YAPI_SUCCESS` and `errmsg` contains the error message.

[Tilt FindTilt]

The `[Tilt FindTilt]` function allows you to find a tilt sensor from the serial number of the module on which it resides and from its function name. You can use logical names as well, as long as you have initialized them. Let us imagine a Yocto-3D-V2-C module with serial number `Y3DMK02C-123456` which you have named `"MyModule"`, and for which you have given the `tilt1` function the name `"MyFunction"`. The following five calls are strictly equivalent, as long as `"MyFunction"` is defined only once.

```
YTilt *tilt = [Tilt FindTilt:@"Y3DMK02C-123456.tilt1"];
YTilt *tilt = [Tilt FindTilt:@"Y3DMK02C-123456.MyFunction"];
YTilt *tilt = [Tilt FindTilt:@"MyModule.tilt1"];
YTilt *tilt = [Tilt FindTilt:@"MyModule.MyFunction"];
YTilt *tilt = [Tilt FindTilt:@"MyFunction"];
```

`[Tilt FindTilt]` returns an object which you can then use at will to control the tilt sensor.

isOnline

The `isOnline` method of the object returned by `[Tilt FindTilt]` allows you to know if the corresponding module is present and in working order.

get_currentValue

The `get_currentValue()` method of the object returned by `YTilt.FindTilt` provides the value of orientation measured by the tilt sensor. The value returned is a floating number.

YCompass.FindCompass, YGyro.FindGyro...

Functions `YCompass.FindCompass`, `YMagnetometer.FindMagnetometer`, `YGyro.FindGyro` and `YAccelerometer.FindAccelerometer` allow you to work with compass, magnetometer, gyroscope, acceleration measures. You can handle them just as `YTilt.FindTilt`.

A real example

Launch Xcode 4.2 and open the corresponding sample project provided in the directory **Examples/Doc-GettingStarted-Yocto-3D-V2-C** of the Yoctopuce library.

In this example, you will recognize the functions explained above, but this time used with all side materials needed to make it work nicely as a small demo.

```
#import <Foundation/Foundation.h>
#import "yocto_api.h"
#import "yocto_tilt.h"
#import "yocto_compass.h"
#import "yocto_gyro.h"
#import "yocto_accelerometer.h"

static void usage(void)
{
```

```

NSLog(@"usage: demo <serial_number> ");
NSLog(@"      demo <logical_name>");
NSLog(@"      demo any          (use any discovered device)");
exit(1);
}

int main(int argc, const char * argv[])
{
    NSError *error;

    if (argc < 2) {
        usage();
    }

    @autoreleasepool {
        // Setup the API to use local USB devices
        if([YAPI RegisterHub:@"usb": &error] != YAPI_SUCCESS) {
            NSLog(@"RegisterHub error: %@", [error localizedDescription]);
            return 1;
        }

        NSString *target = [NSString stringWithUTF8String:argv[1]];
        YTilt *anytilt, *tilt1, *tilt2;
        YCompass *compass;
        YAccelerometer *accelerometer;
        YGyro *gyro;

        if([target isEqualToString:@"any"]) {
            anytilt = [YTilt FirstTilt];
            if (anytilt == NULL) {
                NSLog(@"No module connected (check USB cable)");
                return 1;
            }
        } else {
            anytilt = [YTilt FindTilt:[target stringByAppendingString:@"tilt1"]];
            if(![anytilt isOnline]) {
                NSLog(@"Module not connected (check identification and USB cable)");
                return 1;
            }
        }

        NSString *serial = [[anytilt get_module] get_serialNumber];
        // retrieve all sensors on the device matching the serial
        tilt1 = [YTilt FindTilt:[serial stringByAppendingString:@"tilt1"]];
        tilt2 = [YTilt FindTilt:[serial stringByAppendingString:@"tilt2"]];
        compass = [YCompass FindCompass:[serial stringByAppendingString:@"compass"]];
        accelerometer = [YAccelerometer FindAccelerometer:[serial stringByAppendingString:@"accelerometer"]];
        gyro = [YGyro FindGyro:[serial stringByAppendingString:@"gyro"]];
        int count = 0;

        while(1) {
            if(![tilt1 isOnline]) {
                NSLog(@"device disconnected");
                break;
            }
            if ((count % 10) == 0)
                NSLog(@"tilt1\ntilt2\ncompass\tacc\tgyro");
            NSLog(@"%f\n%f\n%f\t%f\t%f",
                [tilt1 get_currentValue],
                [tilt2 get_currentValue],
                [compass get_currentValue],
                [accelerometer get_currentValue],
                [gyro get_currentValue]);
            count++;

            [YAPI Sleep:250:NULL];
        }
        [YAPI FreeAPI];
    }

    return 0;
}

```

20.2. Control of the module part

Each module can be controlled in a similar manner, you can find below a simple sample program displaying the main parameters of the module and enabling you to activate the localization beacon.

```
#import <Foundation/Foundation.h>
#import "yocto_api.h"

static void usage(const char *exe)
{
    NSLog(@"usage: %s <serial or logical name> [ON/OFF]\n", exe);
    exit(1);
}

int main (int argc, const char * argv[])
{
    NSError *error;

    @autoreleasepool {
        // Setup the API to use local USB devices
        if([YAPI RegisterHub:@"usb": &error] != YAPI_SUCCESS) {
            NSLog(@"RegisterHub error: %@", [error localizedDescription]);
            return 1;
        }
        if(argc < 2)
            usage(argv[0]);
        NSString *serial_or_name = [NSString stringWithUTF8String:argv[1]];
        // use serial or logical name
        YModule *module = [YModule FindModule:serial_or_name];
        if ([module isOnline]) {
            if (argc > 2) {
                if (strcmp(argv[2], "ON") == 0)
                    [module setBeacon:Y_BEACON_ON];
                else
                    [module setBeacon:Y_BEACON_OFF];
            }
            NSLog(@"serial:      %@\n", [module serialNumber]);
            NSLog(@"logical name: %@\n", [module logicalName]);
            NSLog(@"luminosity:   %d\n", [module luminosity]);
            NSLog(@"beacon:      ");
            if ([module beacon] == Y_BEACON_ON)
                NSLog(@"ON\n");
            else
                NSLog(@"OFF\n");
            NSLog(@"upTime:      %ld sec\n", [module upTime] / 1000);
            NSLog(@"USB current: %d mA\n", [module usbCurrent]);
            NSLog(@"logs:       %@\n", [module get_lastLogs]);
        } else {
            NSLog(@"%@ not connected (check identification and USB cable)\n",
                serial_or_name);
        }
        [YAPI FreeAPI];
    }
    return 0;
}
```

Each property `xxx` of the module can be read thanks to a method of type `get_xxxx`, and properties which are not read-only can be modified with the help of the `set_xxx` method. For more details regarding the used functions, refer to the API chapters.

Changing the module settings

When you want to modify the settings of a module, you only need to call the corresponding `set xxx` function. However, this modification is performed only in the random access memory (RAM) of the module: if the module is restarted, the modifications are lost. To memorize them persistently, it is necessary to ask the module to save its current configuration in its permanent memory. To do so, use the `saveToFlash` method. Inversely, it is possible to force the module to forget its current settings by using the `revertFromFlash` method. The short example below allows you to modify the logical name of a module.

```

#import <Foundation/Foundation.h>
#import "yocto_api.h"

static void usage(const char *exe)
{
    NSLog(@"usage: %s <serial> <newLogicalName>\n", exe);
    exit(1);
}

int main (int argc, const char * argv[])
{
    NSError *error;

    @autoreleasepool {
        // Setup the API to use local USB devices
        if([YAPI RegisterHub:@"usb" :&error] != YAPI_SUCCESS) {
            NSLog(@"RegisterHub error: %@", [error localizedDescription]);
            return 1;
        }

        if(argc < 2)
            usage(argv[0]);

        NSString *serial_or_name = [NSString stringWithUTF8String:argv[1]];
        // use serial or logical name
        YModule *module = [YModule FindModule:serial_or_name];

        if (module.isOnline) {
            if (argc >= 3) {
                NSString *newname = [NSString stringWithUTF8String:argv[2]];
                if (![YAPI CheckLogicalName:newname]) {
                    NSLog(@"Invalid name (%@)\n", newname);
                    usage(argv[0]);
                }
                module.logicalName = newname;
                [module saveToFlash];
            }
            NSLog(@"Current name: %@\n", module.logicalName);
        } else {
            NSLog(@"%% not connected (check identification and USB cable)\n",
                serial_or_name);
        }
        [YAPI FreeAPI];
    }
    return 0;
}

```

Warning: the number of write cycles of the nonvolatile memory of the module is limited. When this limit is reached, nothing guaranties that the saving process is performed correctly. This limit, linked to the technology employed by the module micro-processor, is located at about 100000 cycles. In short, you can use the `saveToFlash` function only 100000 times in the life of the module. Make sure you do not call this function within a loop.

Listing the modules

Obtaining the list of the connected modules is performed with the `yFirstModule()` function which returns the first module found. Then, you only need to call the `nextModule()` function of this object to find the following modules, and this as long as the returned value is not `NULL`. Below a short example listing the connected modules.

```

#import <Foundation/Foundation.h>
#import "yocto_api.h"

int main (int argc, const char * argv[])
{
    NSError *error;

    @autoreleasepool {
        // Setup the API to use local USB devices
        if([YAPI RegisterHub:@"usb" :&error] != YAPI_SUCCESS) {
            NSLog(@"RegisterHub error: %@\n", [error localizedDescription]);
            return 1;
        }
    }
}

```

```

    NSLog(@"Device list:\n");

    YModule *module = [YModule FirstModule];
    while (module != nil) {
        NSLog(@"%@ %@", module.serialNumber, module.productName);
        module = [module nextModule];
    }
    [YAPI FreeAPI];
}
return 0;
}

```

20.3. Error handling

When you implement a program which must interact with USB modules, you cannot disregard error handling. Inevitably, there will be a time when a user will have unplugged the device, either before running the software, or even while the software is running. The Yoctopuce library is designed to help you support this kind of behavior, but your code must nevertheless be conceived to interpret in the best possible way the errors indicated by the library.

The simplest way to work around the problem is the one used in the short examples provided in this chapter: before accessing a module, check that it is online with the `isOnline` function, and then hope that it will stay so during the fraction of a second necessary for the following code lines to run. This method is not perfect, but it can be sufficient in some cases. You must however be aware that you cannot completely exclude an error which would occur after the call to `isOnline` and which could crash the software. The only way to prevent this is to implement one of the two error handling techniques described below.

The method recommended by most programming languages for unpredictable error handling is the use of exceptions. By default, it is the behavior of the Yoctopuce library. If an error happens while you try to access a module, the library throws an exception. In this case, there are three possibilities:

- If your code catches the exception and handles it, everything goes well.
- If your program is running in debug mode, you can relatively easily determine where the problem happened and view the explanatory message linked to the exception.
- Otherwise... the exception makes your program crash, bang!

As this latest situation is not the most desirable, the Yoctopuce library offers another possibility for error handling, allowing you to create a robust program without needing to catch exceptions at every line of code. You simply need to call the `YAPI.DisableExceptions()` function to commute the library to a mode where exceptions for all the functions are systematically replaced by specific return values, which can be tested by the caller when necessary. For each function, the name of each return value in case of error is systematically documented in the library reference. The name always follows the same logic: a `get_state()` method returns a `ClassName.STATE_INVALID` value, a `get_currentValue` method returns a `ClassName.CURRENTVALUE_INVALID` value, and so on. In any case, the returned value is of the expected type and is not a null pointer which would risk crashing your program. At worst, if you display the value without testing it, it will be outside the expected bounds for the returned value. In the case of functions which do not normally return information, the return value is `YAPI_SUCCESS` if everything went well, and a different error code in case of failure.

When you work without exceptions, you can obtain an error code and an error message explaining the source of the error. You can request them from the object which returned the error, calling the `errType()` and `errMessage()` methods. Their returned values contain the same information as in the exceptions when they are active.

21. Using with unsupported languages

Yoctopuce modules can be driven from most common programming languages. New languages are regularly added, depending on the interest expressed by Yoctopuce product users. Nevertheless, some languages are not, and will never be, supported by Yoctopuce. There can be several reasons for this: compilers which are not available anymore, unadapted environments, and so on.

However, there are alternative methods to access Yoctopuce modules from an unsupported programming language.

21.1. Command line

The easiest method to drive Yoctopuce modules from an unsupported programming language is to use the command line API through system calls. The command line API is in fact made of a group of small executables which are easy to call. Their output is also easy to analyze. As most programming languages allow you to make system calls, the issue is solved with a few lines of code.

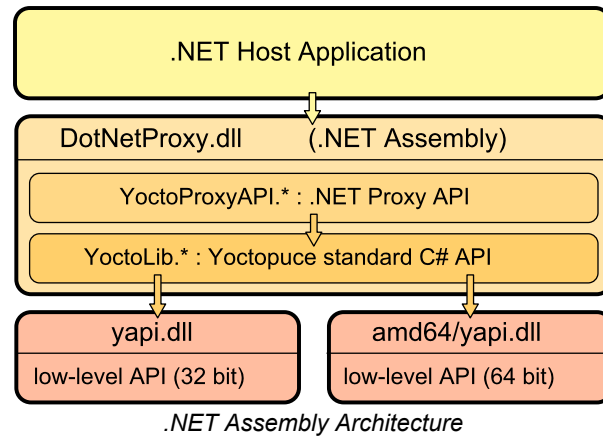
However, if the command line API is the easiest solution, it is neither the fastest nor the most efficient. For each call, the executable must initialize its own API and make an inventory of USB connected modules. This requires about one second per call.

21.2. .NET Assembly

A .NET Assembly enables you to share a set of pre-compiled classes to offer a service, by stating entry points which can be used by third-party applications. In our case, it's the whole Yoctopuce library which is available in the .NET Assembly, so that it can be used in any environment which supports .NET Assembly dynamic loading.

The Yoctopuce library as a .NET Assembly does not contain only the standard C# Yoctopuce library, as this would not have allowed an optimal use in all environments. Indeed, we cannot expect host applications to necessarily offer a thread system or a callback system, although they are very useful to manage plug-and-play events and sensors with a high refresh rate. Likewise, we cannot expect from external applications a transparent behavior in cases where a function call in Assembly creates a delay because of network communications.

Therefore, we added to it an additional layer, called *.NET Proxy* library. This additional layer offers an interface very similar to the standard library but somewhat simplified, as it internally manages all the callback mechanisms. Instead, this library offers mirror objects, called *Proxys*, which publish through *Properties* the main attributes of the Yoctopuce functions such as the current measure, configuration parameters, the state, and so on.



The callback mechanism automatically updates the properties of the *Proxys* objects, without the host application needing to care for it. The later can thus, at any time and without any risk of latency, display the value of all properties of Yoctopuce Proxy objects.

Pay attention to the fact that the `yapi.dll` low-level communication library is **not** included in the .NET Assembly. You must therefore keep it together with `DotNetProxyLibrary.dll`. The 32 bit version must be located in the same directory as `DotNetProxyLibrary.dll`, while the 64 bit version must be in a subdirectory `amd64`.

Example of use with MATLAB

Here is how to load our Proxy .NET Assembly in MATLAB and how to read the value of the first sensor connected by USB found on the machine:

```

NET.addAssembly("C:/Yoctopuce/DotNetProxyLibrary.dll");
import YoctoProxyAPI.*

errmsg = YAPIProxy.RegisterHub("usb");
sensor = YSensorProxy.FindSensor("");
measure = sprintf('%0.3f %s', sensor.CurrentValue, sensor.Unit);

```

Example of use in PowerShell

PowerShell commands are a little stranger, but we can recognize the same structure:

```

Add-Type -Path "C:/Yoctopuce/DotNetProxyLibrary.dll"

$errmsg = [YoctoProxyAPI.YAPIProxy]::RegisterHub("usb")
$sensor = [YoctoProxyAPI.YSensorProxy]::FindSensor("")
$measure = "{0:n3} {1}" -f $sensor.CurrentValue, $sensor.Unit

```

Specificities of the .NET Proxy library

With regards to classic Yoctopuce libraries, the following differences in particular should be noted:

No FirstModule/nextModule method

To obtain an object referring to the first found module, we call `YModuleProxy.FindModule("")`. If there is no connected module, this method returns an object with its `module.IsOnline` property set to `False`. As soon as a module is connected, the property changes to `True` and the module hardware identifier is updated.

To list modules, you can call the `module.GetSimilarFunctions()` method which returns an array of character strings containing the identifiers of all the modules which were found.

No callback function

Callback functions are implemented internally and they update the object properties. You can therefore simply poll the properties, without significant performance penalties. Be aware that if you

use one of the function that disables callbacks, the automatic refresh of object properties may not work anymore.

A new method `YAPIProxy.GetLog` makes it possible to retrieve low-level debug logs without using callbacks.

Enumerated types

In order to maximize compatibility with host applications, the .NET Proxy library does not use true .NET enumerated types, but simple integers. For each enumerated type, the library includes public constants named according to the possible values. Contrarily to standard Yoctopuce libraries, numeric values always start from 1, as the value 0 is reserved to return an invalid value, for instance when the device is disconnected.

Invalid numeric results

For all numeric results, rather than using an arbitrary constant, the invalid value returned in case of error is *NaN*. You should therefore use function *isNaN()* to detect this value.

Using .NET Assembly without the Proxy library

If for a reason or another you do not want to use the Proxy library, and if your environment allows it, you can use the standard C# API as it is located in the Assembly, under the `YoctoLib` namespace. Beware however not to mix both types of use: either you go through the Proxy library, or you use the `YoctoLib` version directly, but not both!

Compatibility

For the LabVIEW Yoctopuce library to work correctly with your Yoctopuce modules, these modules need to have firmware 37120, or higher.

In order to be compatible with as many versions of Windows as possible, including Windows XP, the *DotNetProxyLibrary.dll* library is compiled in .NET 3.5, which is available by default on all the Windows versions since XP. As of today, we have never met any non-Windows environment able to load a .NET Assembly, so we only ship the low-level communication dll for Windows together with the assembly.

21.3. VirtualHub and HTTP GET

VirtualHub is available on almost all current platforms. It is generally used as a gateway to provide access to Yoctopuce modules from languages which prevent direct access to hardware layers of a computer (JavaScript, PHP, Java, ...).

In fact, VirtualHub is a small web server able to route HTTP requests to Yoctopuce modules. This means that if you can make an HTTP request from your programming language, you can drive Yoctopuce modules, even if this language is not officially supported.

REST interface

At a low level, the modules are driven through a REST API. Thus, to control a module, you only need to perform appropriate requests on the *VirtualHub*. By default, the VirtualHub HTTP port is 4444.

An important advantage of this technique is that preliminary tests are very easy to implement. You only need a VirtualHub and a simple web browser. If you copy the following URL in your preferred browser, while VirtualHub is running, you obtain the list of the connected modules.

```
http://127.0.0.1:4444/api/services/whitePages.txt
```

Note that the result is displayed as text, but if you request *whitePages.xml*, you obtain an XML result. Likewise, *whitePages.json* allows you to obtain a JSON result. The *html* extension even allows you to display a rough interface where you can modify values in real time. The whole REST API is available in these different formats.

Driving a module through the REST interface

Each Yoctopuce module has its own REST interface, available in several variants. Let us imagine a Yocto-3D-V2-C with the *Y3DMK02C-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/Y3DMK02C-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/Y3DMK02C-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/Y3DMK02C-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 tilt1 function, build the following URL:

```
http://127.0.0.1:4444/bySerial/Y3DMK02C-12345/api/tilt1.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/Y3DMK02C-12345/api/tilt1/logicalName
```

Rather logically, attributes can be modified in the same manner.

```
http://127.0.0.1:4444/bySerial/Y3DMK02C-12345/api/tilt1?logicalName=myFunction
```

You can find the list of available attributes for your Yocto-3D-V2-C at the beginning of the *Programming* chapter.

Accessing Yoctopuce data logger through the REST interface

This section only applies to devices with a built-in data logger.

The preview of all recorded data streams can be retrieved in JSON format using the following URL:

```
http://127.0.0.1:4444/bySerial/Y3DMK02C-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/Y3DMK02C-12345/dataLogger.json?id=tilt1&utc=1389801080
```

21.4. Using dynamic libraries

The low level Yoctopuce API is available under several formats of dynamic libraries written in C. The sources are available with the C++ API. If you use one of these low level libraries, you do not need VirtualHub anymore.

Filename	Platform
libyapi.dylib	Max OS X
libyapi-amd64.so	Linux Intel (64 bits)
libyapi-armel.so	Linux ARM EL (32 bits)
libyapi-armhf.so	Linux ARM HL (32 bits)
libyapi-aarch64.so	Linux ARM (64 bits)
libyapi-i386.so	Linux Intel (32 bits)
yapi64.dll	Windows (64 bits)
yapi.dll	Windows (32 bits)

These dynamic libraries contain all the functions necessary to completely rebuild the whole high level API in any language able to integrate these libraries. This chapter nevertheless restrains itself to describing basic use of the modules.

Driving a module

The three essential functions of the low level API are the following:

```
int yapiInitAPI(int connection_type, char *errmsg);
int yapiUpdateDeviceList(int forceupdate, char *errmsg);
int yapiHTTPRequest(char *device, char *request, char* buffer, int buffsize, int *fullsize,
char *errmsg);
```

The *yapiInitAPI* function initializes the API and must be called once at the beginning of the program. For a USB type connection, the *connection_type* parameter takes value 1. The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message. This pointer can also point to *null*. The function returns a negative integer in case of error, zero otherwise.

The *yapiUpdateDeviceList* manages the inventory of connected Yoctopuce modules. It must be called at least once. To manage hot plug and detect potential newly connected modules, this function must be called at regular intervals. The *forceupdate* parameter must take value 1 to force a hardware scan. The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message. This pointer can also point to *null*. The function returns a negative integer in case of error, zero otherwise.

Finally, the *yapiHTTPRequest* function sends HTTP requests to the module REST API. The *device* parameter contains the serial number or the logical name of the module which you want to reach. The *request* parameter contains the full HTTP request (including terminal line breaks). *buffer* points to a character buffer long enough to contain the answer. *buffersize* is the size of the buffer. *fullsize* is a pointer to an integer to which will be assigned the actual size of the answer. The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message. This pointer can also point to *null*. The function returns a negative integer in case of error, zero otherwise.

The format of the requests is the same as the one described in the *VirtualHub et HTTP GET* section. All the character strings used by the API are strings made of 8-bit characters: Unicode and UTF8 are not supported.

The result returned in the buffer variable respects the HTTP protocol. It therefore includes an HTTP header. This header ends with two empty lines, that is a sequence of four ASCII characters 13, 10, 13, 10.

Here is a sample program written in pascal using the *yapi.dll* DLL to read and then update the luminosity of a module.

```
// Dll functions import
function yapiInitAPI(mode:integer;
```

```

errmsg : pansichar):integer;cdecl;
external 'yapi.dll' name 'yapiInitAPI';
function yapiUpdateDeviceList(force:integer;errmsg : pansichar):integer;cdecl;
external 'yapi.dll' name 'yapiUpdateDeviceList';
function yapiHTTPRequest(device:pansichar;url:pansichar; buffer:pansichar;
buffsize:integer;var fullsize:integer;
errmsg : pansichar):integer;cdecl;
external 'yapi.dll' name 'yapiHTTPRequest';

var
errmsgBuffer : array [0..256] of ansichar;
dataBuffer : array [0..1024] of ansichar;
errmsg,data : pansichar;
fullsize,p : integer;

const
serial = 'Y3DMK02C-12345';
getValue = 'GET /api/module/luminosity HTTP/1.1'#13#10#13#10;
setValue = 'GET /api/module?luminosity=100 HTTP/1.1'#13#10#13#10;

begin
errmsg := @errmsgBuffer;
data := @dataBuffer;
// API initialization
if(yapiInitAPI(1,errmsg)<0) then
begin
writeln(errmsg);
halt;
end;

// forces a device inventory
if( yapiUpdateDeviceList(1,errmsg)<0) then
begin
writeln(errmsg);
halt;
end;

// requests the module luminosity
if (yapiHTTPRequest(serial,getValue,data,sizeof(dataBuffer),fullsize,errmsg)<0) then
begin
writeln(errmsg);
halt;
end;

// searches for the HTTP header end
p := pos(#13#10#13#10,data);

// displays the response minus the HTTP header
writeln(copy(data,p+4,length(data)-p-3));

// changes the luminosity
if (yapiHTTPRequest(serial,setValue,data,sizeof(dataBuffer),fullsize,errmsg)<0) then
begin
writeln(errmsg);
halt;
end;

end.

```

Module inventory

To perform an inventory of Yoctopuce modules, you need two functions from the dynamic library:

```

int yapiGetAllDevices(int *buffer,int maxsize,int *neededsize,char *errmsg);
int yapiGetDeviceInfo(int devdesc,yDeviceSt *infos, char *errmsg);

```

The *yapiGetAllDevices* function retrieves the list of all connected modules as a list of handles. *buffer* points to a 32-bit integer array which contains the returned handles. *maxsize* is the size in bytes of the buffer. To *neededsize* is assigned the necessary size to store all the handles. From this, you can deduce either the number of connected modules or that the input buffer is too small. The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message. This pointer can also point to *null*. The function returns a negative integer in case of error, zero otherwise.

The *yapiGetDeviceInfo* function retrieves the information related to a module from its handle. *devdesc* is a 32-bit integer representing the module and which was obtained through *yapiGetAllDevices*. *infos* points to a data structure in which the result is stored. This data structure has the following format:

Name	Type	Size (bytes)	Description
vendorid	int	4	Yoctopuce USB ID
deviceid	int	4	Module USB ID
devrelease	int	4	Module version
nbinbterfaces	int	4	Number of USB interfaces used by the module
manufacturer	char[]	20	Yoctopuce (null terminated)
productname	char[]	28	Model (null terminated)
serial	char[]	20	Serial number (null terminated)
logicalname	char[]	20	Logical name (null terminated)
firmware	char[]	22	Firmware version (null terminated)
beacon	byte	1	Beacon state (0/1)

The *errmsg* parameter must point to a 255 character buffer to retrieve a potential error message.

Here is a sample program written in pascal using the *yapi.dll* DLL to list the connected modules.

```
// device description structure
type yDeviceSt = packed record
  vendorid      : word;
  deviceid      : word;
  devrelease    : word;
  nbinbterfaces : word;
  manufacturer  : array [0..19] of ansichar;
  productname   : array [0..27] of ansichar;
  serial        : array [0..19] of ansichar;
  logicalname   : array [0..19] of ansichar;
  firmware      : array [0..21] of ansichar;
  beacon        : byte;
end;

// Dll function import
function yapiInitAPI(mode:integer;
  errmsg : pansichar):integer;cdecl;
  external 'yapi.dll' name 'yapiInitAPI';

function yapiUpdateDeviceList(force:integer;errmsg : pansichar):integer;cdecl;
  external 'yapi.dll' name 'yapiUpdateDeviceList';

function yapiGetAllDevices( buffer:pointer;
  maxsize:integer;
  var neededsize:integer;
  errmsg : pansichar):integer; cdecl;
  external 'yapi.dll' name 'yapiGetAllDevices';

function apiGetDeviceInfo(d:integer; var infos:yDeviceSt;
  errmsg : pansichar):integer; cdecl;
  external 'yapi.dll' name 'yapiGetDeviceInfo';

var
  errmsgBuffer : array [0..256] of ansichar;
  dataBuffer    : array [0..127] of integer; // max of 128 USB devices
  errmsg,data   : pansichar;
  neededsize,i  : integer;
  devinfos      : yDeviceSt;

begin
  errmsg := @errmsgBuffer;

  // API initialization
  if(yapiInitAPI(1,errmsg)<0) then
  begin
    writeln(errmsg);
    halt;
  end;
```

```
// forces a device inventory
if( yapiUpdateDeviceList(1,errmsg)<0) then
begin
  writeln(errmsg);
  halt;
end;

// loads all device handles into dataBuffer
if yapiGetAllDevices(@dataBuffer,sizeof(dataBuffer),neededsize,errmsg)<0 then
begin
  writeln(errmsg);
  halt;
end;

// gets device info from each handle
for i:=0 to neededsize div sizeof(integer)-1 do
begin
  if (apiGetDeviceInfo(dataBuffer[i], devinfos, errmsg)<0) then
  begin
    writeln(errmsg);
    halt;
  end;
  writeln(pansichar(@devinfos.serial)+' ('+pansichar(@devinfos.productname)+')');
end;

end.
```

VB6 and yapi.dll

Each entry point from the yapi.dll is duplicated. You will find one regular C-decl version and one Visual Basic 6 compatible version, prefixed with *vb6_*.

21.5. Porting the high level library

As all the sources of the Yoctopuce API are fully provided, you can very well port the whole API in the language of your choice. Note, however, that a large portion of the API source code is automatically generated.

Therefore, it is not necessary for you to port the complete API. You only need to port the *yocto_api* file and one file corresponding to a function, for example *yocto_relay*. After a little additional work, Yoctopuce is then able to generate all other files. Therefore, we highly recommend that you contact Yoctopuce support before undertaking to port the Yoctopuce library in another language. Collaborative work is advantageous to both parties.

22. Firmware Update

There are multiples way to update the firmware of a Yoctopuce module.

22.1. VirtualHub or the YoctoHub

It is possible to update the firmware directly from the web interface of VirtualHub or of a YoctoHub. The configuration panel of the module has an "upgrade" button to start a wizard that will guide you through the firmware update procedure.

In case the firmware update fails for any reason, and the module does no start anymore, simply unplug the module then plug it back while maintaining the *Yocto-button* down. The module will boot in "firmware update" mode and will appear in the VirtualHub interface below the module list.

22.2. The command line library

All the command line tools can update Yoctopuce modules thanks to the `downloadAndUpdate` command. The module selection mechanism works like for a traditional command. The `[target]` is the name of the module that you want to update. You can also use the "any" or "all" aliases, or even a name list, where the names are separated by commas, without spaces.

```
C:\>Executable [options] [target] command [parameters]
```

The following example updates all the Yoctopuce modules connected by USB.

```
C:\>YModule all downloadAndUpdate
ok: Yocto-PowerRelay RELAYHI1-266C8(rev=15430) is up to date.
ok: 0 / 0 hubs in 0.000000s.
ok: 0 / 0 shields in 0.000000s.
ok: 1 / 1 devices in 0.130000s 0.130000s per device.
ok: All devices are now up to date.
C:\>
```

22.3. The Android application Yocto-Firmware

You can update your module firmware from your Android phone or tablet with the [Yocto-Firmware](#) application. This application lists all the Yoctopuce modules connected by USB and checks if a more recent firmware is available on www.yoctopuce.com. If a more recent firmware is available, you can

update the module. The application is responsible for downloading and installing the new firmware while preserving the module parameters.

Please note: while the firmware is being updated, the module restarts several times. Android interprets a USB device reboot as a disconnection and reconnection of the USB device and asks the authorization to use the USB port again. The user must click on *OK* for the update process to end successfully.

22.4. Updating the firmware with the programming library

If you need to integrate firmware updates in your application, the libraries offer you an API to update your modules.

Saving and restoring parameters

The `get_allSettings()` method returns a binary buffer enabling you to save a module persistent parameters. This function is very useful to save the network configuration of a YoctoHub for example.

```
YWireless wireless = YWireless.FindWireless("reference");
YModule m = wireless.get_module();
byte[] default_config = m.get_allSettings();
saveFile("default.bin", default_config);
...
```

You can then apply these parameters to other modules with the `set_allSettings()` method.

```
byte[] default_config = loadFile("default.bin");
YModule m = YModule.FirstModule();
while (m != null) {
    if (m.get_productName() == "YoctoHub-Wireless") {
        m.set_allSettings(default_config);
    }
    m = m.next();
}
```

Finding the correct firmware

The first step to update a Yoctopuce module is to find which firmware you must use. The `checkFirmware(path, onlynew)` method of the `YModule` object does exactly this. The method checks that the firmware given as argument (`path`) is compatible with the module. If the `onlynew` parameter is set, this method checks that the firmware is more recent than the version currently used by the module. When the file is not compatible (or if the file is older than the installed version), this method returns an empty string. In the opposite, if the file is valid, the method returns a file access path.

The following piece of code checks that the `c:\tmp\METEOMK1.17328.byn` is compatible with the module stored in the `m` variable.

```
YModule m = YModule.FirstModule();
...
...
string path = "c:\\tmp\\METEOMK1.17328.byn";
string newfirm = m.checkFirmware(path, false);
if (newfirm != "") {
    Console.WriteLine("firmware " + newfirm + " is compatible");
}
...
```

The argument can be a directory (instead of a file). In this case, the method checks all the files of the directory recursively and returns the most recent compatible firmware. The following piece of code checks whether there is a more recent firmware in the `c:\tmp\` directory.

```
YModule m = YModule.FirstModule();
...
...
string path = "c:\\tmp";
string newfirm = m.checkFirmware(path, true);
if (newfirm != "") {
    Console.WriteLine("firmware " + newfirm + " is compatible and newer");
}
...
```

You can also give the "www.yoctopuce.com" string as argument to check whether there is a more recent published firmware on Yoctopuce's web site. In this case, the method returns the firmware URL. You can use this URL to download the firmware on your disk or use this URL when updating the firmware (see below). Obviously, this possibility works only if your machine is connected to Internet.

```
YModule m = YModule.FirstModule();
...
...
string url = m.checkFirmware("www.yoctopuce.com", true);
if (url != "") {
    Console.WriteLine("new firmware is available at " + url );
}
...
```

Updating the firmware

A firmware update can take several minutes. That is why the update process is run as a background task and is driven by the user code thanks to the `YFirmwareUpdate` class.

To update a Yoctopuce module, you must obtain an instance of the `YFirmwareUpdate` class with the `updateFirmware` method of a `YModule` object. The only parameter of this method is the *path* of the firmware that you want to install. This method does not immediately start the update, but returns a `YFirmwareUpdate` object configured to update the module.

```
string newfirm = m.checkFirmware("www.yoctopuce.com", true);
.....
YFirmwareUpdate fw_update = m.updateFirmware(newfirm);
```

The `startUpdate()` method starts the update as a background task. This background task automatically takes care of

1. saving the module parameters
2. restarting the module in "update" mode
3. updating the firmware
4. starting the module with the new firmware version
5. restoring the parameters

The `get_progress()` and `get_progressMessage()` methods enable you to follow the progression of the update. `get_progress()` returns the progression as a percentage (100 = update complete). `get_progressMessage()` returns a character string describing the current operation (deleting, writing, rebooting, ...). If the `get_progress` method returns a negative value, the update process failed. In this case, the `get_progressMessage()` returns an error message.

The following piece of code starts the update and displays the progress on the standard output.

```
YFirmwareUpdate fw_update = m.updateFirmware(newfirm);
.....
int status = fw_update.startUpdate();
while (status < 100 && status >= 0) {
    int newstatus = fw_update.get_progress();
    if (newstatus != status) {
        Console.WriteLine(status + "% "
            + fw_update.get_progressMessage());
    }
}
```

```
YAPI.Sleep(500, ref errmsg);
status = newstatus;
}

if (status < 0) {
    Console.WriteLine("Firmware Update failed: "
        + fw_update.get_progressMessage());
} else {
    Console.WriteLine("Firmware Updated Successfully!");
}
```

An Android characteristic

You can update a module firmware using the Android library. However, for modules connected by USB, Android asks the user to authorize the application to access the USB port.

During firmware update, the module restarts several times. Android interprets a USB device reboot as a disconnection and a reconnection to the USB port, and prevents all USB access as long as the user has not closed the pop-up window. The user has to click on OK for the update process to continue correctly. **You cannot update a module connected by USB to an Android device without having the user interacting with the device.**

22.5. The "update" mode

If you want to erase all the parameters of a module or if your module does not start correctly anymore, you can install a firmware from the "update" mode.

To force the module to work in "update" mode, disconnect it, wait a few seconds, and reconnect it while maintaining the *Yocto-button* down. This will restart the module in "update" mode. This update mode is protected against corruptions and is always available.

In this mode, the module is not detected by the YModule objects anymore. To obtain the list of connected modules in "update" mode, you must use the `YAPI.GetAllBootLoaders()` function. This function returns a character string array with the serial numbers of the modules in "update" mode.

```
List<string> allBootLoader = YAPI.GetAllBootLoaders();
```

The update process is identical to the standard case (see the preceding section), but you must manually instantiate the `YFirmwareUpdate` object instead of calling `module.updateFirmware()`. The constructor takes as argument three parameters: the module serial number, the path of the firmware to be installed, and a byte array with the parameters to be restored at the end of the update (or `null` to restore default parameters).

```
YFirmwareUpdate fw_update;
fw_update = new YFirmwareUpdate(allBootLoader[0], newfirm, null);
int status = fw_update.startUpdate();
.....
```

23. Advanced programming

The preceding chapters have introduced, in each available language, the basic programming functions which can be used with your Yocto-3D-V2-C module. This chapter presents in a more generic manner a more advanced use of your module. Examples are provided in the language which is the most popular among Yoctopuce customers, that is C#. Nevertheless, you can find complete examples illustrating the concepts presented here in the programming libraries of each language.

To remain as concise as possible, examples provided in this chapter do not perform any error handling. Do not copy them "as is" in a production application.

23.1. Event programming

The methods to manage Yoctopuce modules which we presented to you in preceding chapters were polling functions, consisting in permanently asking the API if something had changed. While easy to understand, this programming technique is not the most efficient, nor the most reactive. Therefore, the Yoctopuce programming API also provides an event programming model. This technique consists in asking the API to signal by itself the important changes as soon as they are detected. Each time a key parameter is modified, the API calls a callback function which you have defined in advance.

Detecting module arrival and departure

Hot-plug management is important when you work with USB modules because, sooner or later, you will have to connect or disconnect a module when your application is running. The API is designed to manage module unexpected arrival or departure in a transparent way. But your application must take this into account if it wants to avoid pretending to use a disconnected module.

Event programming is particularly useful to detect module connection/disconnection. Indeed, it is simpler to be told of new connections rather than to have to permanently list the connected modules to deduce which ones just arrived and which ones left. To be warned as soon as a module is connected, you need three pieces of code.

The callback

The callback is the function which is called each time a new Yoctopuce module is connected. It takes as parameter the relevant module.

```
static void deviceArrival(YModule m)
{
    Console.WriteLine("New module : " + m.get_serialNumber());
}
```

Initialization

You must then tell the API that it must call the callback when a new module is connected.

```
YAPI.RegisterDeviceArrivalCallback(deviceArrival);
```

Note that if modules are already connected when the callback is registered, the callback is called for each of the already connected modules.

Triggering callbacks

A classis issue of callback programming is that these callbacks can be triggered at any time, including at times when the main program is not ready to receive them. This can have undesired side effects, such as dead-locks and other race conditions. Therefore, in the Yoctopuce API, module arrival/departure callbacks are called only when the `UpdateDeviceList()` function is running. You only need to call `UpdateDeviceList()` at regular intervals from a timer or from a specific thread to precisely control when the calls to these callbacks happen:

```
// waiting loop managing callbacks
while (true)
{
    // module arrival / departure callback
    YAPI.UpdateDeviceList(ref errmsg);
    // non active waiting time managing other callbacks
    YAPI.Sleep(500, ref errmsg);
}
```

In a similar way, it is possible to have a callback when a module is disconnected. You can find a complete example implemented in your favorite programming language in the *Examples/Prog-EventBased* directory of the corresponding library.

Be aware that in most programming languages, callbacks must be global procedures, and not methods. If you wish for the callback to call the method of an object, define your callback as a global procedure which then calls your method.

Detecting a modification in the value of a sensor

The Yoctopuce API also provides a callback system allowing you to be notified automatically with the value of any sensor, either when the value has changed in a significant way or periodically at a preset frequency. The code necessary to do so is rather similar to the code used to detect when a new module has been connected.

This technique is useful in particular if you want to detect very quick value changes (within a few milliseconds), as it is much more efficient than reading repeatedly the sensor value and therefore gives better performances.

Callback invocation

To enable a better control, value change callbacks are only called when the `YAPI.Sleep()` and `YAPI.HandleEvents()` functions are running. Therefore, you must call one of these functions at a regular interval, either from a timer or from a parallel thread.

```
while (true)
{
    // inactive waiting loop allowing you to trigger
    // value change callbacks
    YAPI.Sleep(500, ref errmsg);
}
```

In programming environments where only the interface thread is allowed to interact with the user, it is often appropriate to call `YAPI.HandleEvents()` from this thread.

The value change callback

This type of callback is called when a magnetometer changes in a significant way. It takes as parameter the relevant function and the new value, as a character string.¹

```
static void valueChangeCallback(YMagnetometer fct, string value)
{
    Console.WriteLine(fct.get_hardwareId() + "=" + value);
}
```

In most programming languages, callbacks are global procedures, not methods. If you wish for the callback to call a method of an object, define your callback as a global procedure which then calls your method. If you need to keep a reference to your object, you can store it directly in the `YMagnetometer` object using function `set_userdata`. You can then retrieve it in the global callback procedure using `get_userdata`.

Setting up a value change callback

The callback is set up for a given Magnetometer function with the help of the `registerValueCallback` method. The following example sets up a callback for the first available Magnetometer function.

```
YMagnetometer f = YMagnetometer.FirstMagnetometer();
f.registerValueCallback(magnetometerChangeCallBack)
```

Note that each module function can thus have its own distinct callback. By the way, if you like to work with value change callbacks, you will appreciate the fact that value change callbacks are not limited to sensors, but are also available for all Yoctopuce devices (for instance, you can also receive a callback any time a relay state changes).

The timed report callback

This type of callback is automatically called at a predefined time interval. The callback frequency can be configured individually for each sensor, with frequencies going from hundred calls per seconds down to one call per hour. The callback takes as parameter the relevant function and the measured value, as an `YMeasure` object. Contrarily to the value change callback that only receives the latest value, an `YMeasure` object provides both minimal, maximal and average values since the timed report callback. Moreover, the measure includes precise timestamps, which makes it possible to use timed reports for a time-based graph even when not handled immediately.

```
static void periodicCallback(YMagnetometer fct, YMeasure measure)
{
    Console.WriteLine(fct.get_hardwareId() + "=" +
        measure.get_averageValue());
}
```

Setting up a timed report callback

The callback is set up for a given Magnetometer function with the help of the `registerTimedReportCallback` method. The callback will only be invoked once a callback frequency as been set using `set_reportFrequency` (which defaults to timed report callback turned off). The frequency is specified as a string (same as for the data logger), by specifying the number of calls per second (/s), per minute (/m) or per hour (/h). The maximal frequency is 100 times per second (i.e. "100/s"), and the minimal frequency is 1 time per hour (i.e. "1/h"). When the frequency is higher than or equal to 1/s, the measure represents an instant value. When the frequency is below, the measure will include distinct minimal, maximal and average values based on a sampling performed automatically by the device.

The following example sets up a timed report callback 4 times per minute for the first available Magnetometer function.

¹ The value passed as parameter is the same as the value returned by the `get_advertisedValue()` method.

```
YMagnetometer f = YMagnetometer.FirstMagnetometer();
f.set_reportFrequency("4/m");
f.registerTimedReportCallback(periodicCallback);
```

As for value change callbacks, each module function can thus have its own distinct timed report callback.

Generic callback functions

It is sometimes desirable to use the same callback function for various types of sensors (e.g. for a generic sensor graphing application). This is possible by defining the callback for an object of class `YSensor` rather than `YMagnetometer`. Thus, the same callback function will be usable with any subclass of `YSensor` (and in particular with `YMagnetometer`). With the callback function, you can use the method `get_unit()` to get the physical unit of the sensor, if you need to display it.

A complete example

You can find a complete example implemented in your favorite programming language in the *Examples/Prog-EventBased* directory of the corresponding library.

23.2. The data logger

Your Yocto-3D-V2-C is equipped with a data logger able to store non-stop the measures performed by the module. The maximal frequency is 100 times per second (i.e. "100/s"), and the minimal frequency is 1 time per hour (i.e. "1/h"). When the frequency is higher than or equal to 1/s, the measure represents an instant value. When the frequency is below, the measure will include distinct minimal, maximal and average values based on a sampling performed automatically by the device.

Note that it is useless and counter-productive to set a recording frequency higher than the native sampling frequency of the recorded sensor.

The data logger flash memory can store about 500'000 instant measures, or 125'000 averaged measures. When the memory is about to be saturated, the oldest measures are automatically erased.

Make sure not to leave the data logger running at high speed unless really needed: the flash memory can only stand a limited number of erase cycles (typically 100'000 cycles). When running at full speed, the datalogger can burn more than 100 cycles per day ! Also be aware that it is useless to record measures at a frequency higher than the refresh frequency of the physical sensor itself.

Starting/stopping the datalogger

The data logger can be started with the `set_recording()` method.

```
YDataLogger l = YDataLogger.FirstDataLogger();
l.set_recording(YDataLogger.RECORDING_ON);
```

It is possible to make the data recording start automatically as soon as the module is powered on.

```
YDataLogger l = YDataLogger.FirstDataLogger();
l.set_autoStart(YDataLogger.AUTOSTART_ON);
l.get_module().saveToFlash(); // do not forget to save the setting
```

Note: Yoctopuce modules do not need an active USB connection to work: they start working as soon as they are powered on. The Yocto-3D-V2-C can store data without necessarily being connected to a computer: you only need to activate the automatic start of the data logger and to power on the module with a simple USB charger.

Erasing the memory

The memory of the data logger can be erased with the `forgetAllDataStreams()` function. Be aware that erasing cannot be undone.

```
YDataLogger l = YDataLogger.FirstDataLogger();
l.forgetAllDataStreams();
```

Choosing the logging frequency

The logging frequency can be set up individually for each sensor, using the method `set_logFrequency()`. The frequency is specified as a string (same as for timed report callbacks), by specifying the number of calls per second (/s), per minute (/m) or per hour (/h). The default value is "1/s".

The following example configures the logging frequency at 15 measures per minute for the first sensor found, whatever its type:

```
YSensor sensor = YSensor.FirstSensor();
sensor.set_logFrequency("15/m");
```

To avoid wasting flash memory, it is possible to disable logging for specified functions. In order to do so, simply use the value "OFF":

```
sensor.set_logFrequency("OFF");
```

Limitation: The Yocto-3D-V2-C cannot use a different frequency for timed-report callbacks and for recording data into the datalogger. You can disable either of them individually, but if you enable both timed-report callbacks and logging for a given function, the two will work at the same frequency.

Retrieving the data

To load recorded measures from the Yocto-3D-V2-C flash memory, you must call the `get_recordedData()` method of the desired sensor, and specify the time interval for which you want to retrieve measures. The time interval is given by the start and stop UNIX timestamp. You can also specify 0 if you don't want any start or stop limit.

The `get_recordedData()` method does not return directly an array of measured values, since in some cases it would cause a huge load that could affect the responsiveness of the application. Instead, this function will return an `YDataSet` object that can be used to retrieve immediately an overview of the measured data (summary), and then to load progressively the details when desired.

Here are the main methods used to retrieve recorded measures:

1. **dataset = sensor.get_recordedData(0,0):** select the desired time interval
2. **dataset.loadMore():** load data from the device, progressively
3. **dataset.get_summary():** get a single measure summarizing the full time interval
4. **dataset.get_preview():** get an array of measures representing a condensed version of the whole set of measures on the selected time interval (reduced by a factor of approx. 200)
5. **dataset.get_measures():** get an array with all detailed measures (that grows while `loadMore` is being called repeatedly)

Measures are instances of `YMeasure`². They store simultaneously the minimal, average and maximal value at a given time, that you can retrieve using methods **get_minValue()**, **get_averageValue()** and **get_maxValue()** respectively. Here is a small example that uses the functions above:

```
// We will retrieve all measures, without time limit
YDataSet dataset = sensor.get_recordedData(0, 0);

// First call to loadMore() loads the summary/preview
dataset.loadMore();
YMeasure summary = dataset.get_summary();
```

² The `YMeasure` objects used by the data logger are exactly the same kind as those passed as argument to the timed report callbacks.

```

string timeFmt = "dd MMM yyyy hh:mm:ss,fff";
string logFmt = "from {0} to {1} : average={2:0.00}{3}";
Console.WriteLine(String.Format(logFmt,
    summary.get_startTimeUTC_asDateTime().ToString(timeFmt),
    summary.get_endTimeUTC_asDateTime().ToString(timeFmt),
    summary.get_averageValue(), sensor.get_unit()));

// Next calls to loadMore() will retrieve measures
Console.WriteLine("loading details");
int progress;
do {
    Console.Write(".");
    progress = dataset.loadMore();
} while(progress < 100);

// All measures have now been loaded
List<YMeasure> details = dataset.get_measures();
foreach (YMeasure m in details) {
    Console.WriteLine(String.Format(logFmt,
        m.get_startTimeUTC_asDateTime().ToString(timeFmt),
        m.get_endTimeUTC_asDateTime().ToString(timeFmt),
        m.get_averageValue(), sensor.get_unit()));
}

```

You will find a complete example demonstrating how to retrieve data from the logger for each programming language directly in the Yoctopuce library. The example can be found in directory *Examples/Prog-DataLogger*.

Timestamp

As the Yocto-3D-V2-C does not have a battery, it cannot guess alone the current time when powered on. Nevertheless, the Yocto-3D-V2-C will automatically try to adjust its real-time reference using the host to which it is connected, in order to properly attach a timestamp to each measure in the datalogger:

- When the Yocto-3D-V2-C is connected to a computer running either the VirtualHub or any application using the Yoctopuce library, it will automatically receive the time from this computer.
- When the Yocto-3D-V2-C is connected to a YoctoHub-Ethernet, it will get the time that the YoctoHub has obtained from the network (using a server from pool.ntp.org)
- When the Yocto-3D-V2-C is connected to a YoctoHub-Wireless, it will get the time provided by the YoctoHub based on its internal battery-powered real-time clock, which was itself configured either from the network or from a computer
- When the Yocto-3D-V2-C is connected to an Android mobile device, it will get the time from the mobile device as long as an app using the Yoctopuce library is launched.

When none of these conditions applies (for instance if the module is simply connected to an USB charger), the Yocto-3D-V2-C will do its best effort to attach a reasonable timestamp to the measures, using the timestamp found on the latest recorded measures. It is therefore possible to "preset to the real time" an autonomous Yocto-3D-V2-C by connecting it to an Android mobile phone, starting the data logger, then connecting the device alone on an USB charger. Nevertheless, be aware that without external time source, the internal clock of the Yocto-3D-V2-C might be subject to a clock skew (theoretically up to 2%).

24. High-level API Reference

This chapter summarizes the high-level API functions to drive your Yocto-3D-V2-C. Syntax and exact type names may vary from one language to another, but, unless otherwise stated, all the functions are available in every language. For detailed information regarding the types of arguments and return values for a given language, refer to the definition file for this language (`yocto_api.*` as well as the other `yocto_*` files that define the function interfaces).

For languages which support exceptions, all of these functions throw exceptions in case of error by default, rather than returning the documented error value for each function. This is by design, to facilitate debugging. It is however possible to disable the use of exceptions using the `yDisableExceptions()` function, in case you prefer to work with functions that return error values.

This chapter does not repeat the programming concepts described earlier, in order to stay as concise as possible. In case of doubt, do not hesitate to go back to the chapter describing in details all configurable attributes.

24.1. Class YAPI

General functions

These general functions should be used to initialize and configure the Yoctopuce library. In most cases, a simple call to function `yRegisterHub()` should be enough. The module-specific functions `yFind...()` or `yFirst...()` should then be used to retrieve an object that provides interaction with the module.

In order to use the functions described here, you should include:

js	<code><script type='text/javascript' src='yocto_api.js'></script></code>
cpp	<code>#include "yocto_api.h"</code>
m	<code>#import "yocto_api.h"</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YModule;</code>
ts	<code>in HTML: import { YAPI, YErrorMsg, YModule, YSensor } from '../dist/esm/yocto_api_browser.js'; in Node.js: import { YAPI, YErrorMsg, YModule, YSensor } from 'yoctolib-cjs/yocto_api_nodejs.js';</code>
es	<code>in HTML: <script src="../lib/yocto_api.js"></script> in node.js: require('yoctolib-es2017/yocto_api.js');</code>

Global functions

YAPI.AddUdevRule(**force**)

Adds a UDEV rule which authorizes all users to access Yoctopuce modules connected to the USB ports.

cpp m uwp ts es

YAPI.CheckLogicalName(**name**)

Checks if a given string is valid as logical name for a module or a function.

cpp m uwp ts es

YAPI.ClearHTTPCallbackCacheDir(**removeFiles**)

Disables the HTTP callback cache.

YAPI.DisableExceptions()

Disables the use of exceptions to report runtime errors.

cpp m uwp ts es

YAPI.EnableExceptions()

Re-enables the use of exceptions for runtime error handling.

cpp m uwp ts es

YAPI.EnableUSBHost(**osContext**)

This function is used only on Android.

YAPI.FreeAPI()

Waits for all pending communications with Yoctopuce devices to be completed then frees dynamically allocated resources used by the Yoctopuce library.

cpp m uwp ts es

YAPI.GetAPIVersion()

Returns the version identifier for the Yoctopuce library in use.

cpp m uwp ts es

YAPI.GetCacheValidity()

Returns the validity period of the data loaded by the library.

cpp

m

uwp

ts

es

YAPI.GetDeviceListValidity()

Returns the delay between each forced enumeration of the used YoctoHubs.

cpp

m

uwp

ts

es

YAPI.GetDllArchitecture()

Returns the system architecture for the Yoctopuce communication library in use.

YAPI.GetDllPath()

Returns the paths of the DLLs for the Yoctopuce library in use.

YAPI.GetLog(lastLogLine)

Retrieves Yoctopuce low-level library diagnostic logs.

YAPI.GetNetworkTimeout()

Returns the network connection delay for `yRegisterHub()` and `yUpdateDeviceList()`.

cpp

m

uwp

ts

es

YAPI.GetTickCount()

Returns the current value of a monotone millisecond-based time counter.

cpp

m

uwp

ts

es

YAPI.HandleEvents(errmsg)

Maintains the device-to-library communication channel.

cpp

m

uwp

ts

es

YAPI.InitAPI(mode, errmsg)

Initializes the Yoctopuce programming library explicitly.

cpp

m

uwp

ts

es

YAPI.PreregisterHub(url, errmsg)

Fault-tolerant alternative to `yRegisterHub()`.

cpp

m

uwp

ts

es

YAPI.RegisterDeviceArrivalCallback(arrivalCallback)

Register a callback function, to be called each time a device is plugged.

cpp

m

uwp

ts

es

YAPI.RegisterDeviceRemovalCallback(removalCallback)

Register a callback function, to be called each time a device is unplugged.

cpp

m

uwp

ts

es

YAPI.RegisterHub(url, errmsg)

Set up the Yoctopuce library to use modules connected on a given machine.

cpp

m

uwp

ts

es

YAPI.RegisterHubDiscoveryCallback(hubDiscoveryCallback)

Register a callback function, to be called each time an Network Hub send an SSDP message.

cpp

m

uwp

ts

es

YAPI.RegisterHubWebsocketCallback(ws, errmsg, authpwd)

Variant to `yRegisterHub()` used to initialize Yoctopuce API on an existing Websocket session, as happens for incoming WebSocket callbacks.

YAPI.RegisterLogFunction(logfun)

Registers a log callback function.

cpp

m

uwp

ts

es

YAPI.SelectArchitecture(*arch*)

Select the architecture or the library to be loaded to access to USB.

YAPI.SetCacheValidity(*cacheValidityMs*)

Change the validity period of the data loaded by the library.

cpp

m

uwp

ts

es

YAPI.SetDelegate(*object*)

(Objective-C only) Register an object that must follow the protocol YDeviceHotPlug.

m

YAPI.SetDeviceListValidity(*deviceListValidity*)

Modifies the delay between each forced enumeration of the used YoctoHubs.

cpp

m

uwp

ts

es

YAPI.SetHTTPCallbackCacheDir(*directory*)

Enables the HTTP callback cache.

YAPI.SetNetworkTimeout(*networkMsTimeout*)

Modifies the network connection delay for yRegisterHub() and yUpdateDeviceList().

cpp

m

uwp

ts

es

YAPI.SetTimeout(*callback*, *ms_timeout*, *args*)

Invoke the specified callback function after a given timeout.

ts

es

YAPI.SetUSBPacketAckMs(*pktAckDelay*)

Enables the acknowledge of every USB packet received by the Yoctopuce library.

YAPI.Sleep(*ms_duration*, *errmsg*)

Pauses the execution flow for a specified duration.

cpp

m

uwp

ts

es

YAPI.TestHub(*url*, *mstimeout*, *errmsg*)

Test if the hub is reachable.

cpp

m

uwp

ts

es

YAPI.TriggerHubDiscovery(*errmsg*)

Force a hub discovery, if a callback as been registered with yRegisterHubDiscoveryCallback it will be called for each net work hub that will respond to the discovery.

cpp

m

uwp

ts

es

YAPI.UnregisterHub(*url*)

Set up the Yoctopuce library to no more use modules connected on a previously registered machine with RegisterHub.

cpp

m

uwp

ts

es

YAPI.UpdateDeviceList(*errmsg*)

Triggers a (re)detection of connected Yoctopuce modules.

cpp

m

uwp

ts

es

YAPI.UpdateDeviceList_async(*callback*, *context*)

Triggers a (re)detection of connected Yoctopuce modules.

24.2. Class YModule

Global parameters control interface for all Yoctopuce devices

The YModule class can be used with all Yoctopuce USB devices. It can be used to control the module global parameters, and to enumerate the functions provided by each module.

In order to use the functions described here, you should include:

js	<code><script type='text/javascript' src='yocto_api.js'></script></code>
cpp	<code>#include "yocto_api.h"</code>
m	<code>#import "yocto_api.h"</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YModule;</code>
ts	<code>in HTML: import { YAPI, YErrorMsg, YModule, YSensor } from '../dist/esm/yocto_api_browser.js'; in Node.js: import { YAPI, YErrorMsg, YModule, YSensor } from 'yoctolib-cjs/yocto_api_nodejs.js';</code>
es	<code>in HTML: <script src='../lib/yocto_api.js'></script> in node.js: require('yoctolib-es2017/yocto_api.js');</code>

Global functions

YModule.FindModule(func)

Allows you to find a module from its serial number or from its logical name.

cpp m uwp ts es

YModule.FindModuleInContext(yctx, func)

Retrieves a module for a given identifier in a YAPI context.

uwp ts es

YModule.FirstModule()

Starts the enumeration of modules currently accessible.

cpp m uwp ts es

YModule methods

module→addFileToHTTPCallback(filename)

Adds a file to the uploaded data at the next HTTP callback.

cpp m uwp ts es

module→checkFirmware(path, onlynew)

Tests whether the byn file is valid for this module.

cpp m uwp ts es

module→clearCache()

Invalidates the cache.

cpp m ts es

module→describe()

Returns a descriptive text that identifies the module.

cpp m ts es

module→download(pathname)

Downloads the specified built-in file and returns a binary buffer with its content.

cpp m uwp ts es

module→functionBaseType(functionIndex)

Retrieves the base type of the *n*th function on the module.

cpp ts es

module→functionCount()

Returns the number of functions (beside the "module" interface) available on the module.

cpp m ts es

module→functionId(functionIndex)

Retrieves the hardware identifier of the *n*th function on the module.

cpp m ts es

module→functionName(functionIndex)

Retrieves the logical name of the *n*th function on the module.

cpp m ts es

module→functionType(functionIndex)

Retrieves the type of the *n*th function on the module.

cpp ts es

module→functionValue(functionIndex)

Retrieves the advertised value of the *n*th function on the module.

cpp m ts es

module→get_allSettings()

Returns all the settings and uploaded files of the module.

cpp m uwp ts es

module→get_beacon()

Returns the state of the localization beacon.

cpp m uwp ts es

module→get_errorMessage()

Returns the error message of the latest error with this module object.

cpp m ts es

module→get_errorType()

Returns the numerical error code of the latest error with this module object.

cpp m ts es

module→get_firmwareRelease()

Returns the version of the firmware embedded in the module.

cpp m uwp ts es

module→get_functionIds(funType)

Retrieve all hardware identifier that match the type passed in argument.

cpp m uwp ts es

module→get_hardwareId()

Returns the unique hardware identifier of the module.

cpp m ts es uwp

module→get_icon2d()

Returns the icon of the module.

cpp m uwp ts es

module→get_lastLogs()

Returns a string with last logs of the module.

cpp m uwp ts es

module→get_logicalName()

Returns the logical name of the module.

cpp m uwp ts es

module→get_luminosity()

Returns the luminosity of the module informative LEDs (from 0 to 100).

cpp m uwp ts es

module→get_parentHub()

Returns the serial number of the YoctoHub on which this module is connected.

cpp m uwp ts es

module→get_persistentSettings()

Returns the current state of persistent module settings.

cpp m uwp ts es

module→get_productId()

Returns the USB device identifier of the module.

cpp m uwp ts es

module→get_productName()

Returns the commercial name of the module, as set by the factory.

cpp m uwp ts es

module→get_productRelease()

Returns the release number of the module hardware, preprogrammed at the factory.

cpp m uwp ts es

module→get_rebootCountdown()

Returns the remaining number of seconds before the module restarts, or zero when no reboot has been scheduled.

cpp m uwp ts es

module→get_serialNumber()

Returns the serial number of the module, as set by the factory.

cpp m uwp ts es

module→get_subDevices()

Returns a list of all the modules that are plugged into the current module.

cpp m uwp ts es

module→get_upTime()

Returns the number of milliseconds spent since the module was powered on.

cpp m uwp ts es

module→get_url()

Returns the URL used to access the module.

cpp m uwp ts es

module→get_usbCurrent()

Returns the current consumed by the module on the USB bus, in milli-amps.

cpp m uwp ts es

module→get_userdata()

Returns the value of the userData attribute, as previously stored using method `set_userdata`.

cpp m ts es

module→get_userVar()

Returns the value previously stored in this attribute.

cpp m uwp ts es

module→hasFunction(funcId)

Tests if the device includes a specific function.

cpp m uwp ts es

module→isOnline()

Checks if the module is currently reachable, without raising any error.

cpp m ts es

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.

cpp m ts es

module→load_async(msValidity, callback, context)

Preloads the module cache with a specified validity duration (asynchronous version).

module→log(text)

Adds a text message to the device logs.

cpp m uwp ts es

module→nextModule()

Continues the module enumeration started using `yFirstModule()`.

cpp m uwp ts es

module→reboot(secBeforeReboot)

Schedules a simple module reboot after the given number of seconds.

cpp m uwp ts es

module→registerBeaconCallback(callback)

Register a callback function, to be called when the localization beacon of the module has been changed.

cpp m uwp ts es

module→registerConfigChangeCallback(callback)

Register a callback function, to be called when a persistent settings in a device configuration has been changed (e.g.

cpp m uwp ts es

module→registerLogCallback(callback)

Registers a device log callback function.

cpp m uwp ts es

module→revertFromFlash()

Reloads the settings stored in the nonvolatile memory, as when the module is powered on.

cpp m uwp ts es

cpp m uwp ts es

cpp m uwp ts es

cpp m uwp ts es

cpp m uwp ts es

cpp m uwp ts es

cpp m ts es

cpp m uwp ts es

cpp m uwp ts es

cpp m uwp ts es

cpp m uwp ts es

cpp m uwp ts es

ts es

24.3. Class YAccelerometer

Accelerometer control interface, available for instance in the Yocto-3D-V2 or the Yocto-Inclinometer

The YAccelerometer class allows you to read and configure Yoctopuce accelerometers. It inherits from YSensor class the core functions to read measurements, to register callback functions, and to access the autonomous datalogger. This class adds the possibility to access x, y and z components of the acceleration vector separately.

In order to use the functions described here, you should include:

es	in HTML: <code><script src="../../lib/yocto_accelerometer.js"></script></code> in node.js: <code>require('yoctolib-es2017/yocto_accelerometer.js');</code>
js	<code><script type='text/javascript' src='yocto_accelerometer.js'></script></code>
cpp	<code>#include "yocto_accelerometer.h"</code>
m	<code>#import "yocto_accelerometer.h"</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YAccelerometer;</code>
ts	in HTML: <code>import { YAccelerometer } from '../../dist/esm/yocto_accelerometer.js';</code> in Node.js: <code>import { YAccelerometer } from 'yoctolib-cjs/yocto_accelerometer.js';</code>

Global functions

YAccelerometer.FindAccelerometer(func)

Retrieves an accelerometer for a given identifier.

cpp m uwp ts es

YAccelerometer.FindAccelerometerInContext(yctx, func)

Retrieves an accelerometer for a given identifier in a YAPI context.

uwp ts es

YAccelerometer.FirstAccelerometer()

Starts the enumeration of accelerometers currently accessible.

cpp m uwp ts es

YAccelerometer.FirstAccelerometerInContext(yctx)

Starts the enumeration of accelerometers currently accessible.

uwp ts es

YAccelerometer.GetSimilarFunctions()

Enumerates all functions of type Accelerometer available on the devices currently reachable by the library, and returns their unique hardware ID.

YAccelerometer methods

accelerometer→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

cpp m uwp ts es

accelerometer→clearCache()

Invalidates the cache.

cpp m ts es

accelerometer→describe()

Returns a short text that describes unambiguously the instance of the accelerometer in the form TYPE(NAME)=SERIAL.FUNCTIONID.

cpp

m

ts

es

accelerometer→get_advMode()

Returns the measuring mode used for the advertised value pushed to the parent hub.

cpp

m

uwp

ts

es

accelerometer→get_advertisedValue()

Returns the current value of the accelerometer (no more than 6 characters).

cpp

m

uwp

ts

es

accelerometer→get_bandwidth()

Returns the measure update frequency, measured in Hz.

cpp

m

uwp

ts

es

accelerometer→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in g, as a floating point number.

cpp

m

uwp

ts

es

accelerometer→get_currentValue()

Returns the current value of the acceleration, in g, as a floating point number.

cpp

m

uwp

ts

es

accelerometer→get_dataLogger()

Returns the `YDataLogger` object of the device hosting the sensor.

cpp

m

uwp

ts

es

accelerometer→get_errorMessage()

Returns the error message of the latest error with the accelerometer.

cpp

m

ts

es

accelerometer→get_errorType()

Returns the numerical error code of the latest error with the accelerometer.

cpp

m

ts

es

accelerometer→get_friendlyName()

Returns a global identifier of the accelerometer in the format `MODULE_NAME.FUNCTION_NAME`.

cpp

m

ts

es

accelerometer→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

cpp

m

ts

es

accelerometer→get_functionId()

Returns the hardware identifier of the accelerometer, without reference to the module.

cpp

m

ts

es

accelerometer→get_hardwareId()

Returns the unique hardware identifier of the accelerometer in the form `SERIAL.FUNCTIONID`.

cpp

m

ts

es

accelerometer→get_highestValue()

Returns the maximal value observed for the acceleration since the device was started.

cpp

m

uwp

ts

es

accelerometer→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

cpp m uwp ts es

accelerometer→get_logicalName()

Returns the logical name of the accelerometer.

cpp m uwp ts es

accelerometer→get_lowestValue()

Returns the minimal value observed for the acceleration since the device was started.

cpp m uwp ts es

accelerometer→get_module()

Gets the YModule object for the device on which the function is located.

cpp m ts es

accelerometer→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

accelerometer→get_recordedData(startTime, endTime)

Retrieves a YDataSet object holding historical data for this sensor, for a specified time interval.

cpp m uwp ts es

accelerometer→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

cpp m uwp ts es

accelerometer→get_resolution()

Returns the resolution of the measured values.

cpp m uwp ts es

accelerometer→get_sensorState()

Returns the sensor state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

cpp m uwp ts es

accelerometer→get_serialNumber()

Returns the serial number of the module, as set by the factory.

cpp m uwp ts es

accelerometer→get_unit()

Returns the measuring unit for the acceleration.

cpp m uwp ts es

accelerometer→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

cpp m ts es

accelerometer→get_xValue()

Returns the X component of the acceleration, as a floating point number.

cpp m uwp ts es

accelerometer→get_yValue()

Returns the Y component of the acceleration, as a floating point number.

cpp m uwp ts es

accelerometer→get_zValue()

Returns the Z component of the acceleration, as a floating point number.

cpp m uwp ts es

accelerometer→isOnline()

Checks if the accelerometer is currently reachable, without raising any error.

cpp m ts es

accelerometer→isOnline_async(callback, context)

Checks if the accelerometer is currently reachable, without raising any error (asynchronous version).

accelerometer→isReadOnly()

Indicates whether changes to the function are prohibited or allowed.

cpp m uwp ts es

accelerometer→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

accelerometer→load(msValidity)

Preloads the accelerometer cache with a specified validity duration.

cpp m ts es

accelerometer→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

cpp m uwp ts es

accelerometer→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

cpp m uwp ts es

accelerometer→load_async(msValidity, callback, context)

Preloads the accelerometer cache with a specified validity duration (asynchronous version).

accelerometer→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

cpp m uwp ts es

accelerometer→nextAccelerometer()

Continues the enumeration of accelerometers started using `yFirstAccelerometer()`.

cpp m uwp ts es

accelerometer→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

cpp m uwp ts es

accelerometer→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

cpp m uwp ts es

accelerometer→set_advMode(newval)

Changes the measuring mode used for the advertised value pushed to the parent hub.

cpp m uwp ts es

accelerometer→set_bandwidth(newval)

Changes the measure update frequency, measured in Hz.

cpp m uwp ts es

accelerometer→set_highestValue(newval)

Changes the recorded maximal value observed.

cpp m uwp ts es

accelerometer→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

cpp m uwp ts es

accelerometer→set_logicalName(newval)

Changes the logical name of the accelerometer.

cpp m uwp ts es

accelerometer→set_lowestValue(newval)

Changes the recorded minimal value observed.

cpp m uwp ts es

accelerometer→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

cpp m uwp ts es

accelerometer→set_resolution(newval)

Changes the resolution of the measured physical values.

cpp m uwp ts es

accelerometer→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

cpp m ts es

accelerometer→startDataLogger()

Starts the data logger on the device.

cpp m uwp ts es

accelerometer→stopDataLogger()

Stops the datalogger on the device.

cpp m uwp ts es

accelerometer→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

cpp m uwp ts es

accelerometer→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

ts es

24.4. Class YMagnetometer

Magnetometer control interface, available for instance in the Yocto-3D-V2

The YSensor class is the parent class for all Yoctopuce sensor types. It can be used to read the current value and unit of any sensor, read the min/max value, configure autonomous recording frequency and access recorded data. It also provides a function to register a callback invoked each time the observed value changes, or at a predefined interval. Using this class rather than a specific subclass makes it possible to create generic applications that work with any Yoctopuce sensor, even those that do not yet exist. Note: The YAnButTon class is the only analog input which does not inherit from YSensor.

In order to use the functions described here, you should include:

es	in HTML: <code><script src="../../lib/yocto_magnetometer.js"></script></code> in node.js: <code>require('yoctolib-es2017/yocto_magnetometer.js');</code>
js	<code><script type='text/javascript' src='yocto_magnetometer.js'></script></code>
cpp	<code>#include "yocto_magnetometer.h"</code>
m	<code>#import "yocto_magnetometer.h"</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YMagnetometer;</code>
ts	in HTML: <code>import { YMagnetometer } from '../../dist/esm/yocto_magnetometer.js';</code> in Node.js: <code>import { YMagnetometer } from 'yoctolib-cjs/yocto_magnetometer.js';</code>

Global functions

YMagnetometer.FindMagnetometer(func)

Retrieves a magnetometer for a given identifier.

cpp m uwp ts es

YMagnetometer.FindMagnetometerInContext(yctx, func)

Retrieves a magnetometer for a given identifier in a YAPI context.

uwp ts es

YMagnetometer.FirstMagnetometer()

Starts the enumeration of magnetometers currently accessible.

cpp m uwp ts es

YMagnetometer.FirstMagnetometerInContext(yctx)

Starts the enumeration of magnetometers currently accessible.

uwp ts es

YMagnetometer.GetSimilarFunctions()

Enumerates all functions of type Magnetometer available on the devices currently reachable by the library, and returns their unique hardware ID.

YMagnetometer methods

magnetometer→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

cpp m uwp ts es

magnetometer→clearCache()

Invalidates the cache.

cpp m ts es

magnetometer→describe()

Returns a short text that describes unambiguously the instance of the magnetometer in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

cpp m ts es

magnetometer→get_advMode()

Returns the measuring mode used for the advertised value pushed to the parent hub.

cpp m uwp ts es

magnetometer→get_advertisedValue()

Returns the current value of the magnetometer (no more than 6 characters).

cpp m uwp ts es

magnetometer→get_bandwidth()

Returns the measure update frequency, measured in Hz.

cpp m uwp ts es

magnetometer→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in mT, as a floating point number.

cpp m uwp ts es

magnetometer→get_currentValue()

Returns the current value of the magnetic field, in mT, as a floating point number.

cpp m uwp ts es

magnetometer→get_dataLogger()

Returns the `YDataLogger` object of the device hosting the sensor.

cpp m uwp ts es

magnetometer→get_errorMessage()

Returns the error message of the latest error with the magnetometer.

cpp m ts es

magnetometer→get_errorType()

Returns the numerical error code of the latest error with the magnetometer.

cpp m ts es

magnetometer→get_friendlyName()

Returns a global identifier of the magnetometer in the format `MODULE_NAME.FUNCTION_NAME`.

cpp m ts es

magnetometer→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

cpp m ts es

magnetometer→get_functionId()

Returns the hardware identifier of the magnetometer, without reference to the module.

cpp m ts es

magnetometer→get_hardwareId()

Returns the unique hardware identifier of the magnetometer in the form `SERIAL.FUNCTIONID`.

cpp m ts es

magnetometer→get_highestValue()

Returns the maximal value observed for the magnetic field since the device was started.

cpp m uwp ts es

magnetometer→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

cpp m uwp ts es

magnetometer→get_logicalName()

Returns the logical name of the magnetometer.

cpp m uwp ts es

magnetometer→get_lowestValue()

Returns the minimal value observed for the magnetic field since the device was started.

cpp m uwp ts es

magnetometer→get_module()

Gets the YModule object for the device on which the function is located.

cpp m ts es

magnetometer→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

magnetometer→get_recordedData(startTime, endTime)

Retrieves a YDataSet object holding historical data for this sensor, for a specified time interval.

cpp m uwp ts es

magnetometer→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

cpp m uwp ts es

magnetometer→get_resolution()

Returns the resolution of the measured values.

cpp m uwp ts es

magnetometer→get_sensorState()

Returns the sensor state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

cpp m uwp ts es

magnetometer→get_serialNumber()

Returns the serial number of the module, as set by the factory.

cpp m uwp ts es

magnetometer→get_unit()

Returns the measuring unit for the magnetic field.

cpp m uwp ts es

magnetometer→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

cpp m ts es

magnetometer→get_xValue()

Returns the X component of the magnetic field, as a floating point number.

cpp m uwp ts es

magnetometer→get_yValue()

Returns the Y component of the magnetic field, as a floating point number.

cpp m uwp ts es

magnetometer→get_zValue()

Returns the Z component of the magnetic field, as a floating point number.

cpp m uwp ts es

magnetometer→isOnline()

Checks if the magnetometer is currently reachable, without raising any error.

cpp m ts es

magnetometer→isOnline_async(callback, context)

Checks if the magnetometer is currently reachable, without raising any error (asynchronous version).

magnetometer→isReadOnly()

Indicates whether changes to the function are prohibited or allowed.

cpp m uwp ts es

magnetometer→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

magnetometer→load(msValidity)

Preloads the magnetometer cache with a specified validity duration.

cpp m ts es

magnetometer→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

cpp m uwp ts es

magnetometer→loadCalibrationPoints(rawValues, refValues)Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

cpp m uwp ts es

magnetometer→load_async(msValidity, callback, context)

Preloads the magnetometer cache with a specified validity duration (asynchronous version).

magnetometer→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

cpp m uwp ts es

magnetometer→nextMagnetometer()Continues the enumeration of magnetometers started using `yFirstMagnetometer()`.

cpp m uwp ts es

magnetometer→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

cpp m uwp ts es

magnetometer→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

cpp m uwp ts es

magnetometer→set_advMode(newval)

Changes the measuring mode used for the advertised value pushed to the parent hub.

cpp m uwp ts es

magnetometer→set_bandwidth(newval)

Changes the measure update frequency, measured in Hz.

cpp m uwp ts es

magnetometer→set_highestValue(newval)

Changes the recorded maximal value observed.

cpp m uwp ts es

magnetometer→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

cpp m uwp ts es

magnetometer→set_logicalName(newval)

Changes the logical name of the magnetometer.

cpp m uwp ts es

magnetometer→set_lowestValue(newval)

Changes the recorded minimal value observed.

cpp m uwp ts es

magnetometer→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

cpp m uwp ts es

magnetometer→set_resolution(newval)

Changes the resolution of the measured physical values.

cpp m uwp ts es

magnetometer→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

cpp m ts es

magnetometer→startDataLogger()

Starts the data logger on the device.

cpp m uwp ts es

magnetometer→stopDataLogger()

Stops the datalogger on the device.

cpp m uwp ts es

magnetometer→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

cpp m uwp ts es

magnetometer→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

ts es

24.5. Class YRefFrame

3D reference frame configuration interface, available for instance in the Yocto-3D-V2 or the Yocto-Inclinometer

The YRefFrame class is used to set up the base orientation of the Yoctopuce inertial sensors. Thanks to this, orientation functions relative to the earth surface plane can use the proper reference frame. For some devices, the class also implements a tridimensional sensor calibration process, which can compensate for local variations of standard gravity and improve the precision of the tilt sensors.

In order to use the functions described here, you should include:

es	in HTML: <code><script src="../../lib/yocto_refframe.js"></script></code> in node.js: <code>require('yoctolib-es2017/yocto_refframe.js');</code>
js	<code><script type='text/javascript' src='yocto_refframe.js'></script></code>
cpp	<code>#include "yocto_refframe.h"</code>
m	<code>#import "yocto_refframe.h"</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YRefFrame;</code>
ts	in HTML: <code>import { YRefFrame } from '../../dist/esm/yocto_refframe.js';</code> in Node.js: <code>import { YRefFrame } from 'yoctolib-cjs/yocto_refframe.js';</code>

Global functions

YRefFrame.FindRefFrame(func)

Retrieves a reference frame for a given identifier.

cpp m uwp ts es

YRefFrame.FindRefFrameInContext(yctx, func)

Retrieves a reference frame for a given identifier in a YAPI context.

uwp ts es

YRefFrame.FirstRefFrame()

Starts the enumeration of reference frames currently accessible.

cpp m uwp ts es

YRefFrame.FirstRefFrameInContext(yctx)

Starts the enumeration of reference frames currently accessible.

uwp ts es

YRefFrame.GetSimilarFunctions()

Enumerates all functions of type RefFrame available on the devices currently reachable by the library, and returns their unique hardware ID.

YRefFrame methods

refframe→cancel3DCalibration()

Aborts the sensors tridimensional calibration process et restores normal settings.

cpp m uwp ts es

refframe→clearCache()

Invalidates the cache.

cpp m ts es

refframe→describe()

Returns a short text that describes unambiguously the instance of the reference frame in the form TYPE(NAME)=SERIAL.FUNCTIONID.

cpp

m

ts

es

refframe→get_3DCalibrationHint()

Returns instructions to proceed to the tridimensional calibration initiated with method `start3DCalibration`.

cpp

m

uwp

ts

es

refframe→get_3DCalibrationLogMsg()

Returns the latest log message from the calibration process.

cpp

m

uwp

ts

es

refframe→get_3DCalibrationProgress()

Returns the global process indicator for the tridimensional calibration initiated with method `start3DCalibration`.

cpp

m

uwp

ts

es

refframe→get_3DCalibrationStage()

Returns index of the current stage of the calibration initiated with method `start3DCalibration`.

cpp

m

uwp

ts

es

refframe→get_3DCalibrationStageProgress()

Returns the process indicator for the current stage of the calibration initiated with method `start3DCalibration`.

cpp

m

uwp

ts

es

refframe→get_advertisedValue()

Returns the current value of the reference frame (no more than 6 characters).

cpp

m

uwp

ts

es

refframe→get_bearing()

Returns the reference bearing used by the compass.

cpp

m

uwp

ts

es

refframe→get_calibrationState()

Returns the 3D sensor calibration state (Yocto-3D-V2 only).

cpp

m

uwp

ts

es

refframe→get_errorMessage()

Returns the error message of the latest error with the reference frame.

cpp

m

ts

es

refframe→get_errorType()

Returns the numerical error code of the latest error with the reference frame.

cpp

m

ts

es

refframe→get_friendlyName()

Returns a global identifier of the reference frame in the format `MODULE_NAME . FUNCTION_NAME`.

cpp

m

ts

es

refframe→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

cpp

m

ts

es

refframe→get_functionId()

Returns the hardware identifier of the reference frame, without reference to the module.

cpp m ts es

reiframe→get_fusionMode()

Returns the sensor fusion mode.

cpp m uwp ts es

reiframe→get_hardwareId()

Returns the unique hardware identifier of the reference frame in the form `SERIAL.FUNCTIONID`.

cpp m ts es

reiframe→get_logicalName()

Returns the logical name of the reference frame.

cpp m uwp ts es

reiframe→get_measureQuality()

Returns estimated quality of the orientation (Yocto-3D-V2 only).

cpp m uwp ts es

reiframe→get_module()

Gets the `YModule` object for the device on which the function is located.

cpp m ts es

reiframe→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

reiframe→get_mountOrientation()

Returns the installation orientation of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

cpp m uwp ts es

reiframe→get_mountPosition()

Returns the installation position of the device, as configured in order to define the reference frame for the compass and the pitch/roll tilt sensors.

cpp m uwp ts es

reiframe→get_serialNumber()

Returns the serial number of the module, as set by the factory.

cpp m uwp ts es

reiframe→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

cpp m ts es

reiframe→isOnline()

Checks if the reference frame is currently reachable, without raising any error.

cpp m ts es

reiframe→isOnline_async(callback, context)

Checks if the reference frame is currently reachable, without raising any error (asynchronous version).

reiframe→isReadOnly()

Indicates whether changes to the function are prohibited or allowed.

cpp m uwp ts es

refframe→load(msValidity)

Preloads the reference frame cache with a specified validity duration.

cpp m ts es

refframe→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

cpp m uwp ts es

refframe→load_async(msValidity, callback, context)

Preloads the reference frame cache with a specified validity duration (asynchronous version).

refframe→more3DCalibration()

Continues the sensors tridimensional calibration process previously initiated using method `start3DCalibration`.

cpp m uwp ts es

refframe→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

cpp m uwp ts es

refframe→nextRefFrame()

Continues the enumeration of reference frames started using `yFirstRefFrame()`.

cpp m uwp ts es

refframe→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

cpp m uwp ts es

refframe→save3DCalibration()

Applies the sensors tridimensional calibration parameters that have just been computed.

cpp m uwp ts es

refframe→set_bearing(newval)

Changes the reference bearing used by the compass.

cpp m uwp ts es

refframe→set_fusionMode(newval)

Change the sensor fusion mode.

cpp m uwp ts es

refframe→set_logicalName(newval)

Changes the logical name of the reference frame.

cpp m uwp ts es

refframe→set_mountPosition(position, orientation)

Changes the compass and tilt sensor frame of reference.

cpp m uwp ts es

refframe→set_userData(data)

Stores a user context provided as argument in the `userData` attribute of the function.

cpp m ts es

refframe→start3DCalibration()

Initiates the sensors tridimensional calibration process.

[cpp](#) [m](#) [uwp](#) [ts](#) [es](#)**reframe→unmuteValueCallbacks()**

Re-enables the propagation of every new advertised value to the parent hub.

[cpp](#) [m](#) [uwp](#) [ts](#) [es](#)**reframe→wait_async(callback, context)**

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

[ts](#) [es](#)

24.6. Class YTilt

Tilt sensor control interface, available for instance in the Yocto-3D-V2 or the Yocto-Inclinometer

The YSensor class is the parent class for all Yoctopuce sensor types. It can be used to read the current value and unit of any sensor, read the min/max value, configure autonomous recording frequency and access recorded data. It also provides a function to register a callback invoked each time the observed value changes, or at a predefined interval. Using this class rather than a specific subclass makes it possible to create generic applications that work with any Yoctopuce sensor, even those that do not yet exist. Note: The YAnBut ton class is the only analog input which does not inherit from YSensor.

In order to use the functions described here, you should include:

es	in HTML: <code><script src="../../lib/yocto_tilt.js"></script></code> in node.js: <code>require('yoctolib-es2017/yocto_tilt.js');</code>
js	<code><script type='text/javascript' src='yocto_tilt.js'></script></code>
cpp	<code>#include "yocto_tilt.h"</code>
m	<code>#import "yocto_tilt.h"</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YTilt;</code>
ts	in HTML: <code>import { YTilt } from '../../dist/esm/yocto_tilt.js';</code> in Node.js: <code>import { YTilt } from 'yoctolib-cjs/yocto_tilt.js';</code>

Global functions

YTilt.FindTilt(func)

Retrieves a tilt sensor for a given identifier.

cpp m uwp ts es

YTilt.FindTiltInContext(yctx, func)

Retrieves a tilt sensor for a given identifier in a YAPI context.

uwp ts es

YTilt.FirstTilt()

Starts the enumeration of tilt sensors currently accessible.

cpp m uwp ts es

YTilt.FirstTiltInContext(yctx)

Starts the enumeration of tilt sensors currently accessible.

uwp ts es

YTilt.GetSimilarFunctions()

Enumerates all functions of type Tilt available on the devices currently reachable by the library, and returns their unique hardware ID.

YTilt methods

tilt→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

cpp m uwp ts es

tilt→calibrateToZero()

Performs a zero calibration for the tilt measurement (Yocto-Inclinometer only).

cpp m uwp ts es

tilt→clearCache()

Invalidates the cache.

cpp m ts es

tilt→describe()

Returns a short text that describes unambiguously the instance of the tilt sensor in the form `TYPE(NAME)=SERIAL.FUNCTIONID`.

cpp m ts es

tilt→get_advMode()

Returns the measuring mode used for the advertised value pushed to the parent hub.

cpp m uwp ts es

tilt→get_advertisedValue()

Returns the current value of the tilt sensor (no more than 6 characters).

cpp m uwp ts es

tilt→get_bandwidth()

Returns the measure update frequency, measured in Hz.

cpp m uwp ts es

tilt→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in degrees, as a floating point number.

cpp m uwp ts es

tilt→get_currentValue()

Returns the current value of the inclination, in degrees, as a floating point number.

cpp m uwp ts es

tilt→get_dataLogger()

Returns the `YDataLogger` object of the device hosting the sensor.

cpp m uwp ts es

tilt→get_errorMessage()

Returns the error message of the latest error with the tilt sensor.

cpp m ts es

tilt→get_errorType()

Returns the numerical error code of the latest error with the tilt sensor.

cpp m ts es

tilt→get_friendlyName()

Returns a global identifier of the tilt sensor in the format `MODULE_NAME.FUNCTION_NAME`.

cpp m ts es

tilt→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

cpp m ts es

tilt→get_functionId()

Returns the hardware identifier of the tilt sensor, without reference to the module.

cpp m ts es

tilt→get_hardwareId()

Returns the unique hardware identifier of the tilt sensor in the form `SERIAL . FUNCTIONID`.

cpp m ts es

tilt→get_highestValue()

Returns the maximal value observed for the inclination since the device was started.

cpp m uwp ts es

tilt→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

cpp m uwp ts es

tilt→get_logicalName()

Returns the logical name of the tilt sensor.

cpp m uwp ts es

tilt→get_lowestValue()

Returns the minimal value observed for the inclination since the device was started.

cpp m uwp ts es

tilt→get_module()

Gets the `YModule` object for the device on which the function is located.

cpp m ts es

tilt→get_module_async(callback, context)

Gets the `YModule` object for the device on which the function is located (asynchronous version).

tilt→get_recordedData(startTime, endTime)

Retrieves a `YDataSet` object holding historical data for this sensor, for a specified time interval.

cpp m uwp ts es

tilt→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

cpp m uwp ts es

tilt→get_resolution()

Returns the resolution of the measured values.

cpp m uwp ts es

tilt→get_sensorState()

Returns the sensor state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

cpp m uwp ts es

tilt→get_serialNumber()

Returns the serial number of the module, as set by the factory.

cpp m uwp ts es

tilt→get_unit()

Returns the measuring unit for the inclination.

cpp m uwp ts es

tilt→get_userData()

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

cpp m ts es

tilt→isOnline()

Checks if the tilt sensor is currently reachable, without raising any error.

cpp m ts es

tilt→isOnline_async(callback, context)

Checks if the tilt sensor is currently reachable, without raising any error (asynchronous version).

tilt→isReadOnly()

Indicates whether changes to the function are prohibited or allowed.

cpp m uwp ts es

tilt→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

tilt→load(msValidity)

Preloads the tilt sensor cache with a specified validity duration.

cpp m ts es

tilt→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

cpp m uwp ts es

tilt→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

cpp m uwp ts es

tilt→load_async(msValidity, callback, context)

Preloads the tilt sensor cache with a specified validity duration (asynchronous version).

tilt→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

cpp m uwp ts es

tilt→nextTilt()

Continues the enumeration of tilt sensors started using `yFirstTilt()`.

cpp m uwp ts es

tilt→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

cpp m uwp ts es

tilt→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

cpp m uwp ts es

tilt→restoreZeroCalibration()

Cancels any previous zero calibration for the tilt measurement (Yocto-Inclinometer only).

cpp m uwp ts es

tilt→set_advMode(newval)

Changes the measuring mode used for the advertised value pushed to the parent hub.

cpp m uwp ts es

tilt→set_bandwidth(newval)

Changes the measure update frequency, measured in Hz.

cpp m uwp ts es

tilt→set_highestValue(newval)

Changes the recorded maximal value observed.

cpp m uwp ts es

tilt→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

cpp m uwp ts es

tilt→set_logicalName(newval)

Changes the logical name of the tilt sensor.

cpp m uwp ts es

tilt→set_lowestValue(newval)

Changes the recorded minimal value observed.

cpp m uwp ts es

tilt→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

cpp m uwp ts es

tilt→set_resolution(newval)

Changes the resolution of the measured physical values.

cpp m uwp ts es

tilt→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

cpp m ts es

tilt→startDataLogger()

Starts the data logger on the device.

cpp m uwp ts es

tilt→stopDataLogger()

Stops the datalogger on the device.

cpp m uwp ts es

tilt→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

cpp m uwp ts es

tilt→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

ts es

24.7. Class YCompass

Compass function control interface, available for instance in the Yocto-3D-V2

The YCompass class allows you to read and configure Yoctopuce compass functions. It inherits from YSensor class the core functions to read measurements, to register callback functions, and to access the autonomous datalogger.

In order to use the functions described here, you should include:

es	in HTML: <code><script src="../../../lib/yocto_compass.js"></script></code> in node.js: <code>require('yoctolib-es2017/yocto_compass.js');</code>
js	<code><script type='text/javascript' src='yocto_compass.js'></script></code>
cpp	<code>#include "yocto_compass.h"</code>
m	<code>#import "yocto_compass.h"</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YCompass;</code>
ts	in HTML: <code>import { YCompass } from '../../../dist/esm/yocto_compass.js';</code> in Node.js: <code>import { YCompass } from 'yoctolib-cjs/yocto_compass.js';</code>

Global functions

YCompass.FindCompass(func)

Retrieves a compass function for a given identifier.

cpp m uwp ts es

YCompass.FindCompassInContext(yctx, func)

Retrieves a compass function for a given identifier in a YAPI context.

uwp ts es

YCompass.FirstCompass()

Starts the enumeration of compass functions currently accessible.

cpp m uwp ts es

YCompass.FirstCompassInContext(yctx)

Starts the enumeration of compass functions currently accessible.

uwp ts es

YCompass.GetSimilarFunctions()

Enumerates all functions of type Compass available on the devices currently reachable by the library, and returns their unique hardware ID.

YCompass methods

compass→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

cpp m uwp ts es

compass→clearCache()

Invalidates the cache.

cpp m ts es

compass→describe()

Returns a short text that describes unambiguously the instance of the compass function in the form TYPE(NAME)=SERIAL.FUNCTIONID.

cpp

m

ts

es

compass→get_advMode()

Returns the measuring mode used for the advertised value pushed to the parent hub.

cpp

m

uwp

ts

es

compass→get_advertisedValue()

Returns the current value of the compass function (no more than 6 characters).

cpp

m

uwp

ts

es

compass→get_bandwidth()

Returns the measure update frequency, measured in Hz.

cpp

m

uwp

ts

es

compass→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in degrees, as a floating point number.

cpp

m

uwp

ts

es

compass→get_currentValue()

Returns the current value of the relative bearing, in degrees, as a floating point number.

cpp

m

uwp

ts

es

compass→get_dataLogger()

Returns the `YDataLogger` object of the device hosting the sensor.

cpp

m

uwp

ts

es

compass→get_errorMessage()

Returns the error message of the latest error with the compass function.

cpp

m

ts

es

compass→get_errorType()

Returns the numerical error code of the latest error with the compass function.

cpp

m

ts

es

compass→get_friendlyName()

Returns a global identifier of the compass function in the format `MODULE_NAME . FUNCTION_NAME`.

cpp

m

ts

es

compass→get_functionDescriptor()

Returns a unique identifier of type `YFUN_DESCR` corresponding to the function.

cpp

m

ts

es

compass→get_functionId()

Returns the hardware identifier of the compass function, without reference to the module.

cpp

m

ts

es

compass→get_hardwareId()

Returns the unique hardware identifier of the compass function in the form `SERIAL . FUNCTIONID`.

cpp

m

ts

es

compass→get_highestValue()

Returns the maximal value observed for the relative bearing since the device was started.

cpp

m

uwp

ts

es

compass→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

cpp m uwp ts es

compass→get_logicalName()

Returns the logical name of the compass function.

cpp m uwp ts es

compass→get_lowestValue()

Returns the minimal value observed for the relative bearing since the device was started.

cpp m uwp ts es

compass→get_magneticHeading()

Returns the magnetic heading, regardless of the configured bearing.

cpp m uwp ts es

compass→get_module()

Gets the YModule object for the device on which the function is located.

cpp m ts es

compass→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

compass→get_recordedData(startTime, endTime)

Retrieves a YDataSet object holding historical data for this sensor, for a specified time interval.

cpp m uwp ts es

compass→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

cpp m uwp ts es

compass→get_resolution()

Returns the resolution of the measured values.

cpp m uwp ts es

compass→get_sensorState()

Returns the sensor state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

cpp m uwp ts es

compass→get_serialNumber()

Returns the serial number of the module, as set by the factory.

cpp m uwp ts es

compass→get_unit()

Returns the measuring unit for the relative bearing.

cpp m uwp ts es

compass→get_userData()

Returns the value of the userData attribute, as previously stored using method set_userData.

cpp m ts es

compass→isOnline()

Checks if the compass function is currently reachable, without raising any error.

cpp m ts es

compass→isOnline_async(callback, context)

Checks if the compass function is currently reachable, without raising any error (asynchronous version).

compass→isReadOnly()

Indicates whether changes to the function are prohibited or allowed.

cpp m uwp ts es

compass→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

compass→load(msValidity)

Preloads the compass function cache with a specified validity duration.

cpp m ts es

compass→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

cpp m uwp ts es

compass→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

cpp m uwp ts es

compass→load_async(msValidity, callback, context)

Preloads the compass function cache with a specified validity duration (asynchronous version).

compass→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

cpp m uwp ts es

compass→nextCompass()

Continues the enumeration of compass functions started using `yFirstCompass()`.

cpp m uwp ts es

compass→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

cpp m uwp ts es

compass→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

cpp m uwp ts es

compass→set_advMode(newval)

Changes the measuring mode used for the advertised value pushed to the parent hub.

cpp m uwp ts es

compass→set_bandwidth(newval)

Changes the measure update frequency, measured in Hz.

cpp m uwp ts es

compass→set_highestValue(newval)

Changes the recorded maximal value observed.

cpp

m

uwp

ts

es

compass→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

cpp

m

uwp

ts

es

compass→set_logicalName(newval)

Changes the logical name of the compass function.

cpp

m

uwp

ts

es

compass→set_lowestValue(newval)

Changes the recorded minimal value observed.

cpp

m

uwp

ts

es

compass→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

cpp

m

uwp

ts

es

compass→set_resolution(newval)

Changes the resolution of the measured physical values.

cpp

m

uwp

ts

es

compass→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

cpp

m

ts

es

compass→startDataLogger()

Starts the data logger on the device.

cpp

m

uwp

ts

es

compass→stopDataLogger()

Stops the datalogger on the device.

cpp

m

uwp

ts

es

compass→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

cpp

m

uwp

ts

es

compass→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

ts

es

24.8. Class YGyro

Gyroscope control interface, available for instance in the Yocto-3D-V2

The YGyro class allows you to read and configure Yoctopuce gyroscopes. It inherits from YSensor class the core functions to read measurements, to register callback functions, and to access the autonomous datalogger. This class adds the possibility to access x, y and z components of the rotation vector separately, as well as the possibility to deal with quaternion-based orientation estimates.

In order to use the functions described here, you should include:

js	<code><script type='text/javascript' src='yocto_gyro.js'></script></code>
cpp	<code>#include "yocto_gyro.h"</code>
m	<code>#import "yocto_gyro.h"</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YGyro;</code>
ts	<code>in HTML: import { YGyro } from '../dist/esm/yocto_gyro.js'; in Node.js: import { YGyro } from 'yoctolib-cjs/yocto_gyro.js';</code>
es	<code>in HTML: <script src='../lib/yocto_gyro.js'></script> in node.js: require('yoctolib-es2017/yocto_gyro.js');</code>

Global functions

YGyro.FindGyro(func)

Retrieves a gyroscope for a given identifier.

cpp m uwp ts es

YGyro.FindGyroInContext(yctx, func)

Retrieves a gyroscope for a given identifier in a YAPI context.

uwp ts es

YGyro.FirstGyro()

Starts the enumeration of gyroscopes currently accessible.

cpp m uwp ts es

YGyro.FirstGyroInContext(yctx)

Starts the enumeration of gyroscopes currently accessible.

uwp ts es

YGyro.GetSimilarFunctions()

Enumerates all functions of type Gyro available on the devices currently reachable by the library, and returns their unique hardware ID.

YGyro methods

gyro→calibrateFromPoints(rawValues, refValues)

Configures error correction data points, in particular to compensate for a possible perturbation of the measure caused by an enclosure.

cpp m uwp ts es

gyro→clearCache()

Invalidates the cache.

cpp m ts es

gyro→describe()

Returns a short text that describes unambiguously the instance of the gyroscope in the form TYPE(NAME)=SERIAL.FUNCTIONID.

cpp m ts es

gyro→get_advMode()

Returns the measuring mode used for the advertised value pushed to the parent hub.

cpp m uwp ts es

gyro→get_advertisedValue()

Returns the current value of the gyroscope (no more than 6 characters).

cpp m uwp ts es

gyro→get_bandwidth()

Returns the measure update frequency, measured in Hz.

cpp m uwp ts es

gyro→get_currentRawValue()

Returns the uncalibrated, unrounded raw value returned by the sensor, in degrees per second, as a floating point number.

cpp m uwp ts es

gyro→get_currentValue()

Returns the current value of the angular velocity, in degrees per second, as a floating point number.

cpp m uwp ts es

gyro→get_dataLogger()

Returns the YDataLogger object of the device hosting the sensor.

cpp m uwp ts es

gyro→get_errorMessage()

Returns the error message of the latest error with the gyroscope.

cpp m ts es

gyro→get_errorType()

Returns the numerical error code of the latest error with the gyroscope.

cpp m ts es

gyro→get_friendlyName()

Returns a global identifier of the gyroscope in the format MODULE_NAME . FUNCTION_NAME.

cpp m ts es

gyro→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

cpp m ts es

gyro→get_functionId()

Returns the hardware identifier of the gyroscope, without reference to the module.

cpp m ts es

gyro→get_hardwareId()

Returns the unique hardware identifier of the gyroscope in the form SERIAL . FUNCTIONID.

cpp m ts es

gyro→get_heading()

Returns the estimated heading angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

cpp

m

uwp

ts

es

gyro→get_highestValue()

Returns the maximal value observed for the angular velocity since the device was started.

cpp

m

uwp

ts

es

gyro→get_logFrequency()

Returns the datalogger recording frequency for this function, or "OFF" when measures are not stored in the data logger flash memory.

cpp

m

uwp

ts

es

gyro→get_logicalName()

Returns the logical name of the gyroscope.

cpp

m

uwp

ts

es

gyro→get_lowestValue()

Returns the minimal value observed for the angular velocity since the device was started.

cpp

m

uwp

ts

es

gyro→get_module()

Gets the YModule object for the device on which the function is located.

cpp

m

ts

es

gyro→get_module_async(callback, context)

Gets the YModule object for the device on which the function is located (asynchronous version).

gyro→get_pitch()

Returns the estimated pitch angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

cpp

m

uwp

ts

es

gyro→get_quaternionW()

Returns the W component (real part) of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

cpp

m

uwp

ts

es

gyro→get_quaternionX()

Returns the X component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

cpp

m

uwp

ts

es

gyro→get_quaternionY()

Returns the Y component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

cpp

m

uwp

ts

es

gyro→get_quaternionZ()

Returns the Z component of the quaternion describing the device estimated orientation, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

cpp

m

uwp

ts

es

gyro→get_recordedData(startTime, endTime)

Retrieves a YDataSet object holding historical data for this sensor, for a specified time interval.

cpp

m

uwp

ts

es

gyro→get_reportFrequency()

Returns the timed value notification frequency, or "OFF" if timed value notifications are disabled for this function.

cpp m uwp ts es

gyro→get_resolution()

Returns the resolution of the measured values.

cpp m uwp ts es

gyro→get_roll()

Returns the estimated roll angle, based on the integration of gyroscopic measures combined with acceleration and magnetic field measurements.

cpp m uwp ts es

gyro→get_sensorState()

Returns the sensor state code, which is zero when there is an up-to-date measure available or a positive code if the sensor is not able to provide a measure right now.

cpp m uwp ts es

gyro→get_serialNumber()

Returns the serial number of the module, as set by the factory.

cpp m uwp ts es

gyro→get_unit()

Returns the measuring unit for the angular velocity.

cpp m uwp ts es

gyro→get_userData()

Returns the value of the userData attribute, as previously stored using method `set_userData`.

cpp m ts es

gyro→get_xValue()

Returns the angular velocity around the X axis of the device, as a floating point number.

cpp m uwp ts es

gyro→get_yValue()

Returns the angular velocity around the Y axis of the device, as a floating point number.

cpp m uwp ts es

gyro→get_zValue()

Returns the angular velocity around the Z axis of the device, as a floating point number.

cpp m uwp ts es

gyro→isOnline()

Checks if the gyroscope is currently reachable, without raising any error.

cpp m ts es

gyro→isOnline_async(callback, context)

Checks if the gyroscope is currently reachable, without raising any error (asynchronous version).

gyro→isReadOnly()

Indicates whether changes to the function are prohibited or allowed.

cpp m uwp ts es

gyro→isSensorReady()

Checks if the sensor is currently able to provide an up-to-date measure.

gyro→load(msValidity)

Preloads the gyroscope cache with a specified validity duration.

cpp m ts es

gyro→loadAttribute(attrName)

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

cpp m uwp ts es

gyro→loadCalibrationPoints(rawValues, refValues)

Retrieves error correction data points previously entered using the method `calibrateFromPoints`.

cpp m uwp ts es

gyro→load_async(msValidity, callback, context)

Preloads the gyroscope cache with a specified validity duration (asynchronous version).

gyro→muteValueCallbacks()

Disables the propagation of every new advertised value to the parent hub.

cpp m uwp ts es

gyro→nextGyro()

Continues the enumeration of gyroscopes started using `yFirstGyro()`.

cpp m uwp ts es

gyro→registerAnglesCallback(callback)

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

cpp m uwp ts es

gyro→registerQuaternionCallback(callback)

Registers a callback function that will be invoked each time that the estimated device orientation has changed.

cpp m uwp ts es

gyro→registerTimedReportCallback(callback)

Registers the callback function that is invoked on every periodic timed notification.

cpp m uwp ts es

gyro→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

cpp m uwp ts es

gyro→set_advMode(newval)

Changes the measuring mode used for the advertised value pushed to the parent hub.

cpp m uwp ts es

gyro→set_bandwidth(newval)

Changes the measure update frequency, measured in Hz.

cpp m uwp ts es

gyro→set_highestValue(newval)

Changes the recorded maximal value observed.

cpp m uwp ts es

gyro→set_logFrequency(newval)

Changes the datalogger recording frequency for this function.

cpp m uwp ts es

gyro→set_logicalName(newval)

Changes the logical name of the gyroscope.

cpp m uwp ts es

gyro→set_lowestValue(newval)

Changes the recorded minimal value observed.

cpp m uwp ts es

gyro→set_reportFrequency(newval)

Changes the timed value notification frequency for this function.

cpp m uwp ts es

gyro→set_resolution(newval)

Changes the resolution of the measured physical values.

cpp m uwp ts es

gyro→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

cpp m ts es

gyro→startDataLogger()

Starts the data logger on the device.

cpp m uwp ts es

gyro→stopDataLogger()

Stops the datalogger on the device.

cpp m uwp ts es

gyro→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

cpp m uwp ts es

gyro→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

ts es

24.9. Class YDataLogger

DataLogger control interface, available on most Yoctopuce sensors.

A non-volatile memory for storing ongoing measured data is available on most Yoctopuce sensors. Recording can happen automatically, without requiring a permanent connection to a computer. The YDataLogger class controls the global parameters of the internal data logger. Recording control (start/stop) as well as data retrieval is done at sensor objects level.

In order to use the functions described here, you should include:

js	<code><script type='text/javascript' src='yocto_module.js'></script></code>
cpp	<code>#include "yocto_module.h"</code>
m	<code>#import "yocto_module.h"</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YDataLogger;</code>
ts	<code>in HTML: import { YDataLogger } from '../dist/esm/yocto_module.js'; in Node.js: import { YDataLogger } from 'yoctolib-cjs/yocto_module.js';</code>
es	<code>in HTML: <script src='../lib/yocto_module.js'></script> in node.js: require('yoctolib-es2017/yocto_module.js');</code>

Global functions

YDataLogger.FindDataLogger(func)

Retrieves a data logger for a given identifier.

cpp m uwp ts es

YDataLogger.FindDataLoggerInContext(yctx, func)

Retrieves a data logger for a given identifier in a YAPI context.

uwp ts es

YDataLogger.FirstDataLogger()

Starts the enumeration of data loggers currently accessible.

cpp m uwp ts es

YDataLogger.FirstDataLoggerInContext(yctx)

Starts the enumeration of data loggers currently accessible.

uwp ts es

YDataLogger.GetSimilarFunctions()

Enumerates all functions of type DataLogger available on the devices currently reachable by the library, and returns their unique hardware ID.

YDataLogger methods

datalogger→clearCache()

Invalidates the cache.

cpp m ts es

datalogger→describe()

Returns a short text that describes unambiguously the instance of the data logger in the form TYPE(NAME)=SERIAL.FUNCTIONID.

cpp m ts es

datalogger→forgetAllDataStreams()

Clears the data logger memory and discards all recorded data streams.

cpp m uwp ts es

dataLogger→get_advertisedValue()

Returns the current value of the data logger (no more than 6 characters).

cpp m uwp ts es

dataLogger→get_autoStart()

Returns the default activation state of the data logger on power up.

cpp m uwp ts es

dataLogger→get_beaconDriven()

Returns true if the data logger is synchronised with the localization beacon.

cpp m uwp ts es

dataLogger→get_currentRunIndex()

Returns the current run number, corresponding to the number of times the module was powered on with the dataLogger enabled at some point.

cpp m uwp ts es

dataLogger→get_dataSets()

Returns a list of YDataSet objects that can be used to retrieve all measures stored by the data logger.

cpp m uwp ts es

dataLogger→get_dataStreams(v)

Builds a list of all data streams hold by the data logger (legacy method).

dataLogger→get_errorMessage()

Returns the error message of the latest error with the data logger.

cpp m ts es

dataLogger→get_errorType()

Returns the numerical error code of the latest error with the data logger.

cpp m ts es

dataLogger→get_friendlyName()

Returns a global identifier of the data logger in the format MODULE_NAME . FUNCTION_NAME.

cpp m ts es

dataLogger→get_functionDescriptor()

Returns a unique identifier of type YFUN_DESCR corresponding to the function.

cpp m ts es

dataLogger→get_functionId()

Returns the hardware identifier of the data logger, without reference to the module.

cpp m ts es

dataLogger→get_hardwareId()

Returns the unique hardware identifier of the data logger in the form SERIAL . FUNCTIONID.

cpp m ts es

dataLogger→get_logicalName()

Returns the logical name of the data logger.

cpp m uwp ts es

dataLogger→get_module()

Gets the `YModule` object for the device on which the function is located.

cpp m ts es

`datalogger→get_module_async(callback, context)`

Gets the `YModule` object for the device on which the function is located (asynchronous version).

`datalogger→get_recording()`

Returns the current activation state of the data logger.

cpp m uwp ts es

`datalogger→get_serialNumber()`

Returns the serial number of the module, as set by the factory.

cpp m uwp ts es

`datalogger→get_timeUTC()`

Returns the Unix timestamp for current UTC time, if known.

cpp m uwp ts es

`datalogger→get_usage()`

Returns the percentage of datalogger memory in use.

cpp m uwp ts es

`datalogger→get_userData()`

Returns the value of the `userData` attribute, as previously stored using method `set_userData`.

cpp m ts es

`datalogger→isOnline()`

Checks if the data logger is currently reachable, without raising any error.

cpp m ts es

`datalogger→isOnline_async(callback, context)`

Checks if the data logger is currently reachable, without raising any error (asynchronous version).

`datalogger→isReadOnly()`

Indicates whether changes to the function are prohibited or allowed.

cpp m uwp ts es

`datalogger→load(msValidity)`

Preloads the data logger cache with a specified validity duration.

cpp m ts es

`datalogger→loadAttribute(attrName)`

Returns the current value of a single function attribute, as a text string, as quickly as possible but without using the cached value.

cpp m uwp ts es

`datalogger→load_async(msValidity, callback, context)`

Preloads the data logger cache with a specified validity duration (asynchronous version).

`datalogger→muteValueCallbacks()`

Disables the propagation of every new advertised value to the parent hub.

cpp m uwp ts es

`datalogger→nextDataLogger()`

Continues the enumeration of data loggers started using `yFirstDataLogger()`.

cpp m uwp ts es

datalogger→registerValueCallback(callback)

Registers the callback function that is invoked on every change of advertised value.

cpp m uwp ts es

datalogger→set_autoStart(newval)

Changes the default activation state of the data logger on power up.

cpp m uwp ts es

datalogger→set_beaconDriven(newval)

Changes the type of synchronisation of the data logger.

cpp m uwp ts es

datalogger→set_logicalName(newval)

Changes the logical name of the data logger.

cpp m uwp ts es

datalogger→set_recording(newval)

Changes the activation state of the data logger to start/stop recording data.

cpp m uwp ts es

datalogger→set_timeUTC(newval)

Changes the current UTC time reference used for recorded data.

cpp m uwp ts es

datalogger→set_userData(data)

Stores a user context provided as argument in the userData attribute of the function.

cpp m ts es

datalogger→unmuteValueCallbacks()

Re-enables the propagation of every new advertised value to the parent hub.

cpp m uwp ts es

datalogger→wait_async(callback, context)

Waits for all pending asynchronous commands on the module to complete, and invoke the user-provided callback function.

ts es

24.10. Class YDataSet

Recorded data sequence, as returned by `sensor.get_recordedData()`

`YDataSet` objects make it possible to retrieve a set of recorded measures for a given sensor and a specified time interval. They can be used to load data points with a progress report. When the `YDataSet` object is instantiated by the `sensor.get_recordedData()` function, no data is yet loaded from the module. It is only when the `loadMore()` method is called over and over than data will be effectively loaded from the `dataLogger`.

A preview of available measures is available using the function `get_preview()` as soon as `loadMore()` has been called once. Measures themselves are available using function `get_measures()` when loaded by subsequent calls to `loadMore()`.

This class can only be used on devices that use a relatively recent firmware, as `YDataSet` objects are not supported by firmwares older than version 13000.

In order to use the functions described here, you should include:

js	<code><script type='text/javascript' src='yocto_module.js'></script></code>
cpp	<code>#include "yocto_module.h"</code>
m	<code>#import "yocto_module.h"</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YDataSet;</code>
ts	<code>in HTML: import { YDataSet } from '../dist/esm/yocto_module.js'; in Node.js: import { YDataSet } from 'yoctolib-cjs/yocto_module.js';</code>
es	<code>in HTML: <script src='../lib/yocto_module.js'></script> in node.js: require('yoctolib-es2017/yocto_module.js');</code>

Global functions

`YDataSet.Init(sensorName, startTime, endTime)`

Retrieves a `YDataSet` object holding historical data for a sensor given by its name or hardware identifier, for a specified time interval.

YDataSet methods

`dataset→get_endTimeUTC()`

Returns the end time of the dataset, relative to the Jan 1, 1970.

cpp m uwp ts es

`dataset→get_functionId()`

Returns the hardware identifier of the function that performed the measure, without reference to the module.

cpp m uwp ts es

`dataset→get_hardwareId()`

Returns the unique hardware identifier of the function who performed the measures, in the form `SERIAL.FUNCTIONID`.

cpp m uwp ts es

`dataset→get_measures()`

Returns all measured values currently available for this `DataSet`, as a list of `YMeasure` objects.

cpp m uwp ts es

`dataset→get_measuresAt(measure)`

Returns the detailed set of measures for the time interval corresponding to a given condensed measures previously returned by `get_preview()`.

[cpp](#) [m](#) [uwp](#) [ts](#) [es](#)
dataset→get_measuresAvgAt(index)

Returns the average value observed during the time interval covered by the specified entry in the preview.

dataset→get_measuresEndTimeAt(index)

Returns the end time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

dataset→get_measuresMaxAt(index)

Returns the largest value observed during the time interval covered by the specified entry in the preview.

dataset→get_measuresMinAt(index)

Returns the smallest value observed during the time interval covered by the specified entry in the preview.

dataset→get_measuresRecordCount()

Returns the number of measurements currently loaded for this data set.

dataset→get_measuresStartTimeAt(index)

Returns the start time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

dataset→get_preview()

Returns a condensed version of the measures that can be retrieved in this `YDataSet`, as a list of `YMeasure` objects.

[cpp](#) [m](#) [uwp](#) [ts](#) [es](#)
dataset→get_previewAvgAt(index)

Returns the average value observed during the time interval covered by the specified entry in the preview.

dataset→get_previewEndTimeAt(index)

Returns the end time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

dataset→get_previewMaxAt(index)

Returns the largest value observed during the time interval covered by the specified entry in the preview.

dataset→get_previewMinAt(index)

Returns the smallest value observed during the time interval covered by the specified entry in the preview.

dataset→get_previewRecordCount()

Returns the number of entries in the preview summarizing this data set

dataset→get_previewStartTimeAt(index)

Returns the start time of the specified entry in the preview, relative to the Jan 1, 1970 UTC (Unix timestamp).

dataset→get_progress()

Returns the progress of the downloads of the measures from the data logger, on a scale from 0 to 100.

[cpp](#) [m](#) [uwp](#) [ts](#) [es](#)
dataset→get_startTimeUTC()

Returns the start time of the dataset, relative to the Jan 1, 1970.

[cpp](#) [m](#) [uwp](#) [ts](#) [es](#)
dataset→get_summary()

Returns an `YMeasure` object which summarizes the whole `YDataSet`.

[cpp](#) [m](#) [uwp](#) [ts](#) [es](#)
dataset→get_summaryAvg()

Returns the average value observed during the time interval covered by this data set.

dataset→get_summaryEndTime()

Returns the end time of the last measure in the data set, relative to the Jan 1, 1970 UTC (Unix timestamp).

dataset→get_summaryMax()

Returns the largest value observed during the time interval covered by this data set.

dataset→get_summaryMin()

Returns the smallest value observed during the time interval covered by this data set.

dataset→get_summaryStartTime()

Returns the start time of the first measure in the data set, relative to the Jan 1, 1970 UTC (Unix timestamp).

dataset→get_unit()

Returns the measuring unit for the measured value.

`cpp` `m` `uwp` `ts` `es`

dataset→loadMore()

Loads the next block of measures from the dataLogger, and updates the progress indicator.

`cpp` `m` `uwp` `ts` `es`

dataset→loadMore_async(callback, context)

Loads the next block of measures from the dataLogger asynchronously.

24.11. Class YMeasure

Measured value, returned in particular by the methods of the `YDataSet` class.

`YMeasure` objects are used within the API to represent a value measured at a specified time. These objects are used in particular in conjunction with the `YDataSet` class, but also for sensors periodic timed reports (see `sensor.registerTimedReportCallback`).

In order to use the functions described here, you should include:

js	<code><script type='text/javascript' src='yocto_module.js'></script></code>
cpp	<code>#include "yocto_module.h"</code>
m	<code>#import "yocto_module.h"</code>
uwp	<code>import com.yoctopuce.YoctoAPI.YMeasure;</code>
ts	<code>in HTML: import { YMeasure } from '../dist/esm/yocto_module.js'; in Node.js: import { YMeasure } from 'yoctolib-cjs/yocto_module.js';</code>
es	<code>in HTML: <script src="../lib/yocto_module.js"></script> in node.js: require('yoctolib-es2017/yocto_module.js');</code>

YMeasure methods

`measure→get_averageValue()`

Returns the average value observed during the time interval covered by this measure.

cpp m uwp ts es

`measure→get_endTimeUTC()`

Returns the end time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

cpp m uwp ts es

`measure→get_maxValue()`

Returns the largest value observed during the time interval covered by this measure.

cpp m uwp ts es

`measure→get_minValue()`

Returns the smallest value observed during the time interval covered by this measure.

cpp m uwp ts es

`measure→get_startTimeUTC()`

Returns the start time of the measure, relative to the Jan 1, 1970 UTC (Unix timestamp).

cpp m uwp ts es

25. Troubleshooting

25.1. Where to start?

If it is the first time that you use a Yoctopuce module and you do not really know where to start, have a look at the Yoctopuce blog. There is a section dedicated to beginners ¹.

25.2. Programming examples don't seem to work

Most of Yoctopuce API programming examples are command line programs and require some parameters to work properly. You have to start them from your operating system command prompt, or configure your IDE to run them with the proper parameters. ².

25.3. Linux and USB

To work correctly under Linux, the library needs to have write access to all the Yoctopuce USB peripherals. However, by default under Linux, USB privileges of the non-root users are limited to read access. To avoid having to run the library as root, you need to create a new *udev* rule to authorize one or several users to have write access to the Yoctopuce peripherals.

To add a new *udev* rule to your installation, you must add a file with a name following the "`##-arbitraryName.rules`" format, in the `/etc/udev/rules.d` directory. When the system is starting, *udev* reads all the files with a `.rules` extension in this directory, respecting the alphabetical order (for example, the `51-custom.rules` file is interpreted AFTER the `50-udev-default.rules` file).

The `50-udev-default` file contains the system default *udev* rules. To modify the default behavior, you therefore need to create a file with a name that starts with a number larger than 50, that will override the system default rules. Note that to add a rule, you need a root access on the system.

In the `udev_conf` directory of the VirtualHub for Linux³ archive, there are two rule examples which you can use as a basis.

¹ see: http://www.yoctopuce.com/EN/blog_by_categories/for-the-beginners

² see: <http://www.yoctopuce.com/EN/article/about-programming-examples>

³ <http://www.yoctopuce.com/FR/virtualhub.php>

Example 1: 51-yoctopuce.rules

This rule provides all the users with read and write access to the Yoctopuce USB devices. Access rights for all other devices are not modified. If this scenario suits you, you only need to copy the "51-yoctopuce_all.rules" file into the "/etc/udev/rules.d" directory and to restart your system.

```
# udev rules to allow write access to all users
# for Yoctopuce USB devices
SUBSYSTEM=="usb", ATTR{idVendor}=="24e0", MODE="0666"
```

Example 2: 51-yoctopuce_group.rules

This rule authorizes the "yoctogroup" group to have read and write access to Yoctopuce USB peripherals. Access rights for all other peripherals are not modified. If this scenario suits you, you only need to copy the "51-yoctopuce_group.rules" file into the "/etc/udev/rules.d" directory and restart your system.

```
# udev rules to allow write access to all users of "yoctogroup"
# for Yoctopuce USB devices
SUBSYSTEM=="usb", ATTR{idVendor}=="24e0", MODE="0664", GROUP="yoctogroup"
```

25.4. ARM Platforms: HF and EL

There are two main flavors of executable on ARM: HF (Hard Float) binaries, and EL (EABI Little Endian) binaries. These two families are not compatible at all. The compatibility of a given ARM platform with one of these two families depends on the hardware and on the OS build. ArmHL and ArmEL compatibility problems are quite difficult to detect. Most of the time, the OS itself is unable to make a difference between an HF and an EL executable and will return meaningless messages when you try to use the wrong type of binary.

All pre-compiled Yoctopuce binaries are provided in both formats, as two separate ArmHF et ArmEL executables. If you do not know what family your ARM platform belongs to, just try one executable from each family.

25.5. Powered module but invisible for the OS

If your Yocto-3D-V2-C is connected by USB, if its blue led is on, but if the operating system cannot see the module, check that you are using a true USB cable with data wires, and not a charging cable. Charging cables have only power wires.

25.6. Another process named xxx is already using yAPI

If when initializing the Yoctopuce API, you obtain the *"Another process named xxx is already using yAPI"* error message, it means that another application is already using Yoctopuce USB modules. On a single machine only one process can access Yoctopuce modules by USB at a time. You can easily work around this limitation by using VirtualHub and the network mode ⁴.

25.7. Disconnections, erratic behavior

If your Yocto-3D-V2-C behaves erratically and/or disconnects itself from the USB bus without apparent reason, check that it is correctly powered. Avoid cables with a length above 2 meters. If needed, insert a powered USB hub ^{5 6}.

⁴ see: <http://www.yoctopuce.com/EN/article/error-message-another-process-is-already-using-yapi>

⁵ see: <http://www.yoctopuce.com/EN/article/usb-cables-size-matters>

⁶ see: <http://www.yoctopuce.com/EN/article/how-many-usb-devices-can-you-connect>

25.8. After a failed firmware update, the device stopped working

If a firmware update of your Yocto-3D-V2-C fails, it is possible that the module is no longer working. If this is the case, plug in your module while holding down the Yocto-Button. The Yocto-LED should light up brightly and remain steady. Release the button. Your Yocto-3D-V2-C should then appear at the bottom of the VirtualHub user interface as a module waiting to be flashed. This operation also reverts the module to its factory configuration.

25.9. The web interface shows errors after a firmware update

After an update, the windows corresponding to Yocto-3D-V2-C in the VirtualHub user interface report errors. This might be a bug, but it is more likely that your web browser has cached part of the interface code from the previous firmware. Perform a *shift-reload* or clear your browser cache and everything should be back to normal.

25.10. Registering VirtualHub disconnects another instance

If, when performing a call to `RegisterHub()` with a VirtualHub address, another previously registered VirtualHub disconnects, make sure the machine running these VirtualHubs do not have the same *Hostname*. Same *Hostname* can happen very easily when the operating system is installed from a monolithic image, Raspberry Pi are the best example. The Yoctopuce API uses serial numbers to communicate with devices and VirtualHub serial numbers are created on the fly based the *hostname* of the machine running VirtualHub.

25.11. Dropped commands

If, after sending a bunch of commands to a Yoctopuce device, you are under the impression that the last ones have been ignored, a typical example is a quick and dirty program meant to configure a device, make sure you used a `YAPI.FreeAPI()` at the end of the program. Commands are sent to Yoctopuce modules asynchronously thanks to a background thread. When the main program terminates, that thread is killed no matter if some command are left to be sent. However `API.FreeAPI()` waits until there is no more command to send before freeing the API resources and returning.

25.12. Damaged device

Yoctopuce strives to reduce the production of electronic waste. If you believe that your Yocto-3D-V2-C is not working anymore, start by contacting Yoctopuce support by e-mail to diagnose the failure. Even if you know that the device was damaged by mistake, Yoctopuce engineers might be able to repair it, and thus avoid creating electronic waste.



Waste Electrical and Electronic Equipment (WEEE) If you really want to get rid of your Yocto-3D-V2-C, do not throw it away in a trash bin but bring it to your local WEEE recycling point. In this way, it will be disposed properly by a specialized WEEE recycling center.



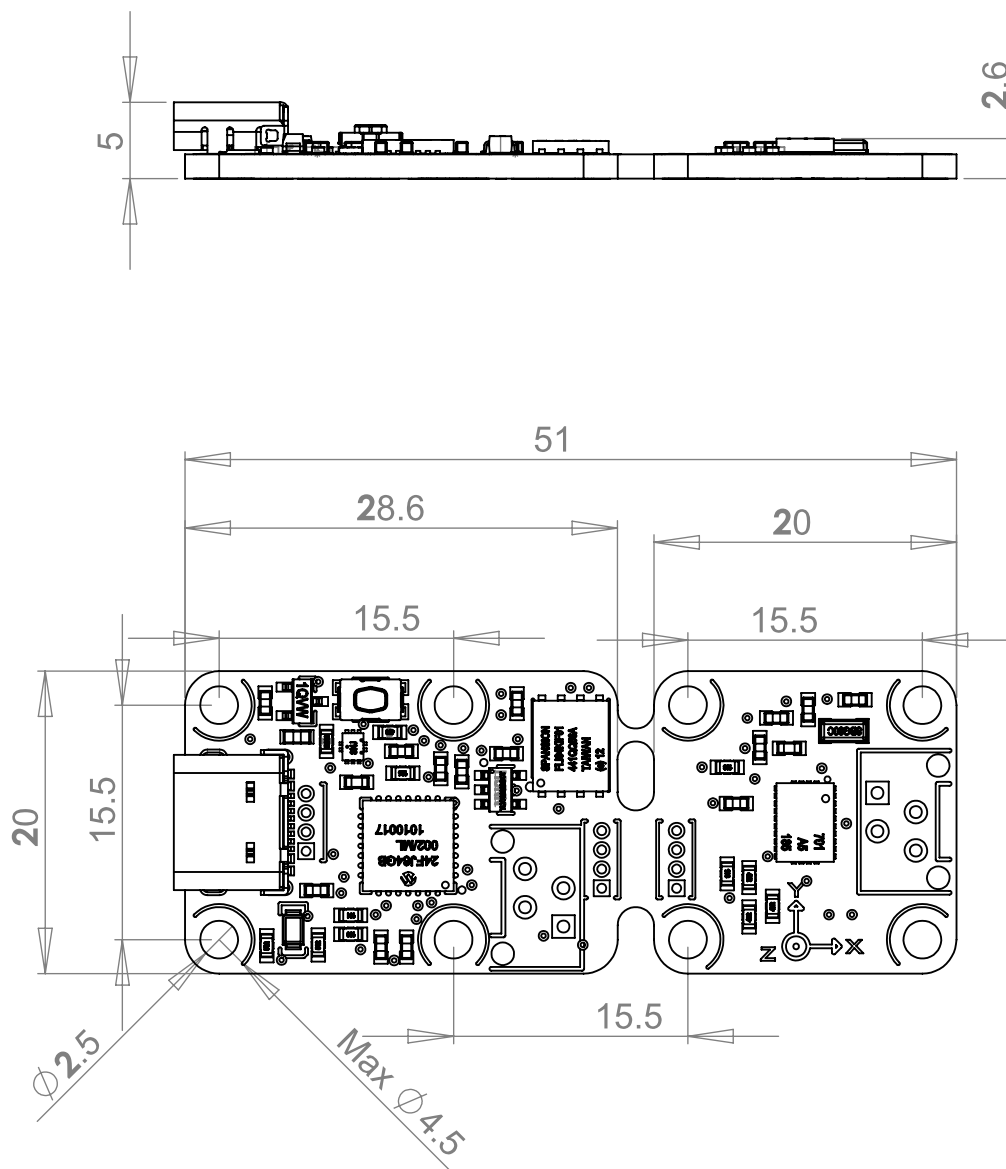
26. Characteristics

You can find below a summary of the main technical characteristics of your Yocto-3D-V2-C module.

Product ID	Y3DMK02C
Hardware release [†]	
USB connector	USB-C
Width	20 mm
Length	51 mm
Weight	4 g
Sensor	BNO055
Gyroscopic attitude estimation	100 Hz
Measuring range (Accel.)	16 g
Measuring range (Gyro)	2000 °/s
Measuring range (Magn.)	13 gauss
Sensitivity	0.1 °
Sensitivity (Accel.)	0.001 g
Sensitivity (Gyro)	0.1 °/s
Sensitivity (Magn.)	0.01 gauss
Protection class, according to IEC 61140	class III
Normal operating temperature	5...40 °C
Extended operating temperature [‡]	-30...85 °C
RoHS compliance	RoHS III (2011/65/UE+2015/863)
USB Vendor ID	0x24E0
USB Device ID	0x00D0
Suggested enclosure	YoctoBox-3D-Black
Harmonized tariff code	9032.9000
Made in	Switzerland

[†] These specifications are for the current hardware revision. Specifications for earlier revisions may differ.

‡ The extended temperature range is defined based on components specifications and has been tested during a limited duration (1h). When using the device in harsh environments for a long period of time, we strongly advise to run extensive tests before going to production.



All dimensions are in mm
Toutes les dimensions sont en mm

Yocto-3D-V2-C

A4

Scale
2:1
Echelle