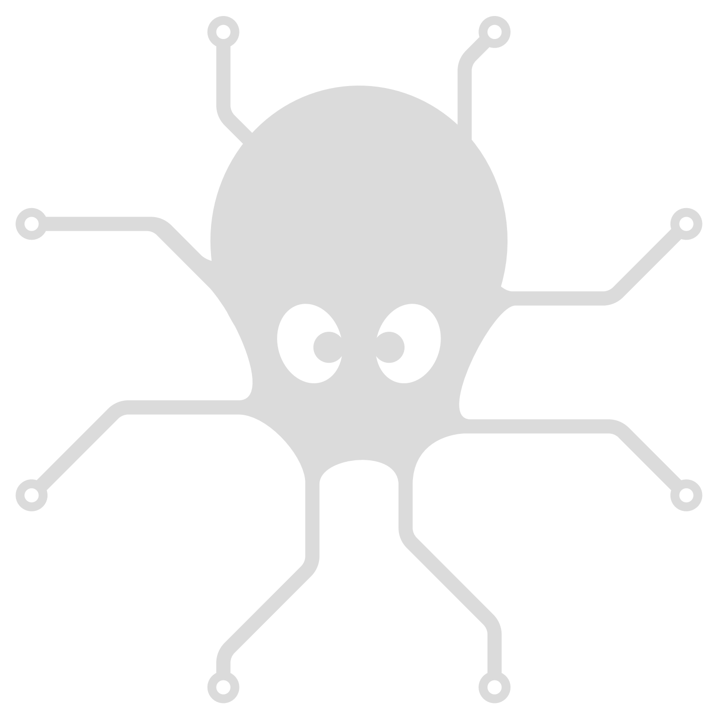


Yocto-SDI12

Mode d'emploi

Table des matières

1. Introduction 1
  1.1. Informations de sécurité 2
  1.2. Conditions environnementales 3

2. Présentation 5
  2.1. Les éléments communs 5
  2.2. Les éléments spécifiques 6
  2.3. Accessoires optionnels 7

3. Premiers pas 9
  3.1. Prérequis 9
  3.2. Test de la connectivité USB 11
  3.3. Localisation 11
  3.4. Test du module 11
  3.5. Configuration 12

4. Montage et connectique 15
  4.1. Fixation 15
  4.2. Contraintes d'alimentation par USB 16
  4.3. Compatibilité électromagnétique (EMI) 16

5. Le port SDI-12 19
  5.1. Fonctionnement d'un bus SDI-12 19
  5.2. Baud Rate et Bytes Frame Format 19
  5.3. Communication entre un maître et un capteur 19
  5.4. Caractères autorisés 20
  5.5. Adresse 20
  5.6. Commandes SDI-12 20
  5.7. Commande de métadonnée 23
  5.8. Utilisation à l'aide des fonctions de l'API 24

6. Mesures automatiques 25
  6.1. Les jobs de communication 25
  6.2. Les tâches 26

6.3. Les commandes 28
6.4. Les fonctions genericSensor 31
6.5. Exemple de configuration 32

7. Programmation, concepts généraux **33**
7.1. Paradigme de programmation 33
7.2. Le module Yocto-SDI12 35
7.3. Module 36
7.4. Sdi12Port 38
7.5. GenericSensor 39
7.6. DataLogger 40
7.7. Files 41
7.8. Quelle interface: Native, DLL ou Service? 42
7.9. Accéder aux modules à travers un hub 44
7.10. Programmation, par où commencer? 45

8. Utilisation du Yocto-SDI12 en ligne de commande **47**
8.1. Installation 47
8.2. Utilisation: description générale 47
8.3. Contrôle de la fonction Sdi12Port 48
8.4. Contrôle de la partie module 49
8.5. Limitations 49

9. Utilisation du Yocto-SDI12 en Python **51**
9.1. Fichiers sources 51
9.2. Librairie dynamique 51
9.3. Contrôle de la fonction Sdi12Port 51
9.4. Contrôle de la partie module 54
9.5. Gestion des erreurs 56

10. Utilisation du Yocto-SDI12 en C++ **57**
10.1. Contrôle de la fonction Sdi12Port 57
10.2. Contrôle de la partie module 60
10.3. Gestion des erreurs 62
10.4. Intégration de la librairie Yoctopuce en C++ 63

11. Utilisation du Yocto-SDI12 en C# **65**
11.1. Installation 65
11.2. Utilisation l'API yoctopuce dans un projet Visual C# 65
11.3. Contrôle de la fonction Sdi12Port 66
11.4. Contrôle de la partie module 68
11.5. Gestion des erreurs 71

12. Utilisation du Yocto-SDI12 avec LabVIEW **73**
12.1. Architecture 73
12.2. Compatibilité 74
12.3. Installation 74
12.4. Présentation des VIs Yoctopuce 79
12.5. Fonctionnement et utilisation des VIs 82
12.6. Utilisation des objets 84
12.7. Gestion du datalogger 86
12.8. Énumération de fonctions 87

12.9. Un mot sur les performances 88
12.10. Un exemple complet de programme LabVIEW 88
12.11. Différences avec les autres API Yoctopuce 89

13. Utilisation du Yocto-SDI12 en Java 91
13.1. Préparation 91
13.2. Contrôle de la fonction Sdi12Port 91
13.3. Contrôle de la partie module 93
13.4. Gestion des erreurs 96

14. Utilisation du Yocto-SDI12 avec Android 97
14.1. Accès Natif et VirtualHub 97
14.2. Préparation 97
14.3. Compatibilité 97
14.4. Activer le port USB sous Android 98
14.5. Contrôle de la fonction Sdi12Port 99
14.6. Contrôle de la partie module 102
14.7. Gestion des erreurs 107

15. Utilisation du Yocto-SDI12 en TypeScript 109
15.1. Utiliser la librairie Yoctopuce pour TypeScript 110
15.2. Petit rappel sur les fonctions asynchrones en JavaScript 110
15.3. Contrôle de la fonction Sdi12Port 111
15.4. Contrôle de la partie module 115
15.5. Gestion des erreurs 117

16. Utilisation du Yocto-SDI12 en JavaScript / EcmaScript 119
16.1. Fonctions bloquantes et fonctions asynchrones en JavaScript 120
16.2. Utiliser la librairie Yoctopuce pour JavaScript / EcmaScript 2017 121
16.3. Contrôle de la fonction Sdi12Port 123
16.4. Contrôle de la partie module 127
16.5. Gestion des erreurs 129

17. Utilisation du Yocto-SDI12 en PHP 131
17.1. Préparation 131
17.2. Contrôle de la fonction Sdi12Port 132
17.3. Contrôle de la partie module 134
17.4. API par callback HTTP et filtres NAT 136
17.5. Gestion des erreurs 140

18. Utilisation du Yocto-SDI12 en VisualBasic .NET 143
18.1. Installation 143
18.2. Utilisation l'API yoctopuce dans un projet Visual Basic 143
18.3. Contrôle de la fonction Sdi12Port 144
18.4. Contrôle de la partie module 146
18.5. Gestion des erreurs 148

19. Utilisation du Yocto-SDI12 en Delphi / Lazarus 151
19.1. Préparation 151
19.2. Contrôle de la fonction Sdi12Port 152
19.3. Contrôle de la partie module 154
19.4. Gestion des erreurs 157

20. Utilisation du Yocto-SDI12 avec Universal Windows Platform 159
20.1. Fonctions bloquantes et fonctions asynchrones 159
20.2. Installation 160
20.3. Utilisation l'API Yoctopuce dans un projet Visual Studio 160
20.4. Contrôle de la fonction Sdi12Port 161
20.5. Un exemple concret 162
20.6. Contrôle de la partie module 163
20.7. Gestion des erreurs 165

21. Utilisation du Yocto-SDI12 en Objective-C 167
21.1. Contrôle de la fonction Sdi12Port 167
21.2. Contrôle de la partie module 169
21.3. Gestion des erreurs 171

22. Utilisation avec des langages non supportés 173
22.1. Utilisation en ligne de commande 173
22.2. Assembly .NET 173
22.3. Virtual Hub et HTTP GET 175
22.4. Utilisation des librairies dynamiques 177
22.5. Port de la librairie haut niveau 180

23. Programmation avancée 181
23.1. Programmation par événements 181
23.2. L'enregistreur de données 184
23.3. Calibration des senseurs 187

24. Mise à jour du firmware 191
24.1. Le VirtualHub ou le YoctoHub 191
24.2. La librairie ligne de commandes 191
24.3. L'application Android Yocto-Firmware 191
24.4. La librairie de programmation 192
24.5. Le mode "mise à jour" 194

25. Référence de l'API de haut niveau 195
25.1. La classe YAPI 196
25.2. La classe YModule 200
25.3. La classe YSdi12Port 207
25.4. La classe YSdi12SensorInfo 215
25.5. La classe YFiles 218
25.6. La classe YGenericSensor 223
25.7. La classe YDataLogger 231
25.8. La classe YDataSet 236
25.9. La classe YMeasure 239

26. Problèmes courants 241
26.1. Par où commencer ? 241
26.2. Linux et USB 241
26.3. Plateformes ARM: HF et EL 242
26.4. Les exemples de programmation n'ont pas l'air de marcher 242
26.5. Module alimenté mais invisible pour l'OS 242
26.6. Another process named xxx is already using yAPI 242

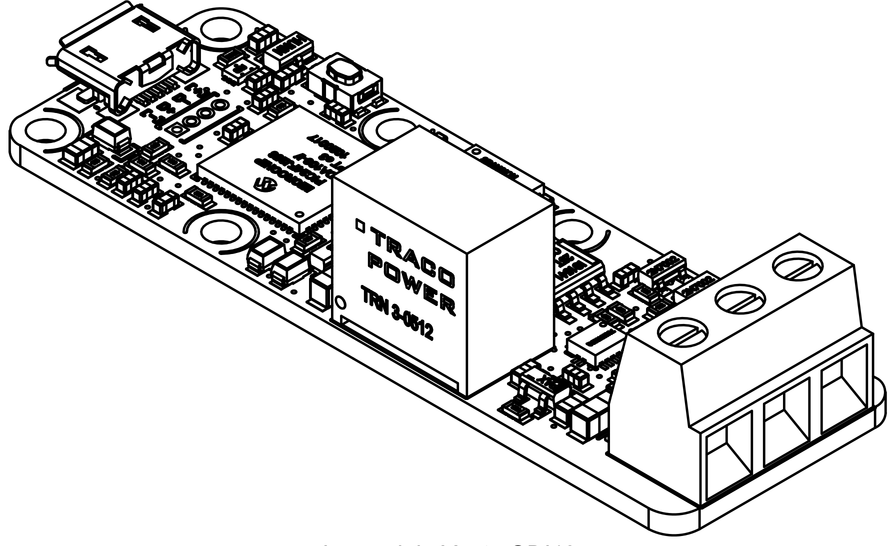
26.7. Déconnexions, comportement erratique 243
26.8. Le module ne marche plus après une mise à jour ratée 243
26.9. RegisterHub d'une instance de VirtualHub déconnecte la précédente 243
26.10. Commandes ignorées 243
26.11. Module endommagé 243

27. Caractéristiques 245
Blueprint 247

1. Introduction

Le module Yocto-SDI12 est un module USB de 60x20mm qui offre la possibilité de communiquer avec des capteurs selon le protocole de communication SDI-12. Il dispose d'une alimentation 12V qui, au besoin, permet d'alimenter les capteurs en question. Le Yocto-SDI12 est essentiellement conçu pour être utilisé comme maître (master) SDI-12, mais peut également être utilisé comme analyseur passif de communications SDI-12.

En plus d'offrir les communications SDI-12 de bas niveau, le Yocto-SDI12 est capable dinterroger et d'analyser de manière autonome l'interface SDI-12 d'un capteur quelconque pour ensuite présenter les résultats à la manière d'un capteur Yoctopuce. En d'autre termes, le Yocto-SDI12 est capable de transformer n'importe quel capteur équipé d'une interface SDI-12 en l'équivalent software d'un capteur Yoctopuce, datalogger compris.



Le module Yocto-SDI12

Le Yocto-SDI12 n'est pas en lui-même un produit complet. C'est un composant destiné à être intégré dans une solution d'automatisation en laboratoire, ou pour le contrôle de procédés industriels, ou pour des applications similaires en milieu résidentiel ou commercial. Pour pouvoir l'utiliser, il faut au minimum l'installer à l'intérieur d'un boîtier de protection et le raccorder à un ordinateur de contrôle.

Yoctopuce vous remercie d'avoir fait l'acquisition de ce Yocto-SDI12 et espère sincèrement qu'il vous donnera entière satisfaction. Les ingénieurs Yoctopuce se sont donné beaucoup de mal pour que votre Yocto-SDI12 soit facile à installer n'importe où et soit facile à piloter depuis un maximum

de langages de programmation. Néanmoins, si ce module venait à vous décevoir, ou si vous avez besoin d'informations supplémentaires, n'hésitez pas à contacter Yoctopuce:

Adresse e-mail:	support@yoctopuce.com
Site Internet:	www.yoctopuce.com
Adresse postale:	Route de Cartigny 33
Localité:	1236 Cartigny
Pays:	Suisse

1.1. Informations de sécurité

Le Yocto-SDI12 est conçu pour respecter la norme de sécurité IEC 61010-1:2010. Il ne causera pas de danger majeur pour l'opérateur et la zone environnante, même en condition de premier défaut, pour autant qu'il soit intégré et utilisé conformément aux instructions contenues dans cette documentation, et en particulier dans cette section.

Boîtier de protection

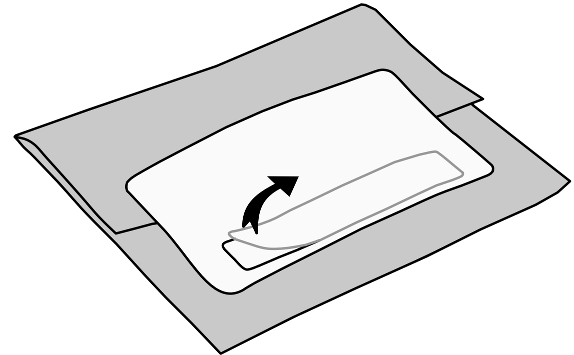
Le Yocto-SDI12 ne doit pas être utilisé sans boîtier de protection, en raison des composants électriques à nu. Pour une sécurité optimale, il devrait être mis dans un boîtier non métallique, non-inflammable, résistant à un choc de 5 J, par exemple en polycarbonate (LEXAN ou autre) d'indice de protection IK08 et classifié V-1 ou mieux selon la norme IEC 60695-11-10. L'utilisation d'un boîtier de qualité inférieure peut nécessiter des avertissements spécifiques pour l'utilisateur et/ou compromettre la conformité avec la norme de sécurité.

Entretien

Si un dégat est constaté sur le circuit électronique ou sur le boîtier, il doit être remplacé afin de ne pas compromettre la sécurité d'utilisation et d'éviter d'endommager d'autres parties du système par les surcharges éventuelles que pourrait causer un court-circuit.

Identification

Pour faciliter l'entretien du circuit et l'identification des risques lors de la maintenance, vous devriez coller l'étiquette autocollante synthétique identifiant le Yocto-SDI12, fournie avec le circuit électronique, à proximité immédiate du module. Si le module est dans un boîtier dédié, l'étiquette devrait être collée sur la surface extérieur du boîtier. L'étiquette est résistante à l'humidité et au frottement usuel qui peut survenir durant un entretien normal.



L'étiquette d'identification est intégrée à l'étiquette de l'emballage.

Applications

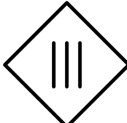
La norme de sécurité vérifiée correspond aux instruments de laboratoire, pour le contrôle de procédés industriels, ou pour des applications similaires en milieu résidentiel ou commercial. Si vous comptez l'utiliser le Yocto-SDI12 pour un autre type d'applications, vous devrez vérifier les critères de conformité en fonction de la norme applicable à votre application.

En particulier, le Yocto-SDI12 n'est *pas* certifié pour utilisation dans un environnement médical, ni pour les applications critiques à la santé, ni pour toute autre application menaçant la vie humaine.

Environnement

Le Yocto-SDI12 n'est *pas* certifié pour utilisation dans les zones dangereuses, ni pour les environnements explosifs. Les conditions environnementales assignées sont décrites ci-dessous.

Classe de protection III (IEC 61140)



Le module Yocto-SDI12 a été conçu pour travailler uniquement avec des très basses tension de sécurité. Ne dépassez pas les tensions indiquées dans ce manuel, et ne raccordez en aucun cas sur le bornier du Yocto-SDI12 un fil susceptible d'être connecté au réseau secteur.

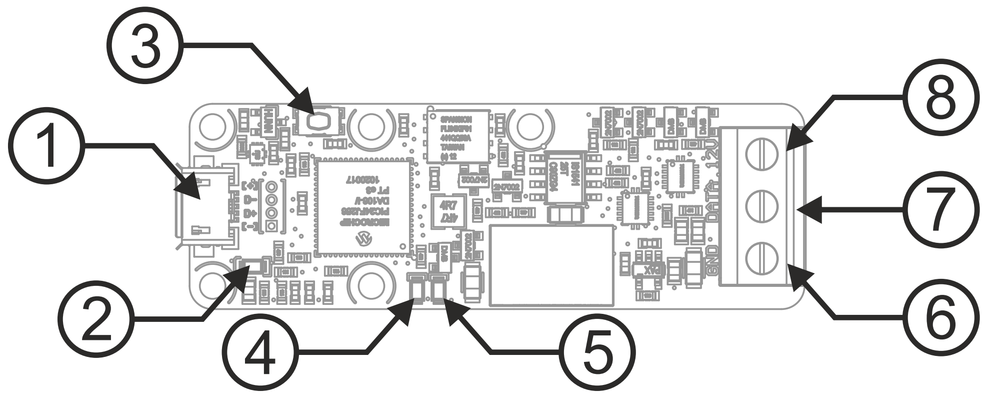
1.2. Conditions environnementales

Les produits Yoctopuce sont conçus pour une utilisation intérieure dans un environnement usuel de bureau ou de laboratoire (*degré de pollution 2* selon IEC 60664): la pollution de l'air doit être faible et essentiellement non conductrice. L'humidité relative prévue est de 10% à 90% RH, sans condensation. L'utilisation dans un environnement avec une pollution solide ou conductrice significative exige de protéger le module contre cette pollution par un boîtier certifié IP67 ou IP68. Les produits sont conçus pour une utilisation jusqu'à une altitude de 2000m.

Le fonctionnement de tous les modules Yoctopuce est garanti conforme à la documentation et aux spécifications de précision pour des conditions de température ambiante normales selon IEC61010-1, soit 5°C à 40°C. De plus, la plupart des modules peuvent aussi être utilisés sur une plage de température étendue, à laquelle quelques limitations peuvent s'appliquer selon les cas.

La plage de température de fonctionnement étendue du Yocto-SDI12 est -30...85°C. Cette plage de température a été déterminée en fonction des recommandations officielles des fabricants des composants utilisés dans le Yocto-SDI12, et par des tests de durée limitée (1h) dans les conditions extrêmes, en environnement controllé. Si vous envisagez d'utiliser le Yocto-SDI12 dans des conditions de température extrêmes pour une période prolongée, il est recommandé de faire des tests extensifs avant la mise en production.

2. Présentation



1: Connecteur USB (micro-B)
2: Yocto-Led
3: Yocto-bouton
4: Led témoin d'émission
5: Led témoin de réception
6: Masse
7: Data
8: Sortie alimentation 12V

2.1. Les éléments communs

Tous les Yocto-modules ont un certain nombre de fonctionnalités en commun.

Le connecteur USB

Les modules de Yoctopuce sont tous équipés d'une connectique USB 2.0 au format micro-B. Attention, le connecteur USB est simplement soudé en surface et peut être arraché si la prise USB venait à faire levier. Si les pistes sont restées en place, le connecteur peut être ressoudé à l'aide d'un bon fer et de flux. Alternativement, vous pouvez souder un fil USB directement dans les trous espacés de 1.27mm prévus à cet effet, prêt du connecteur.

Si vous utilisez une source de tension autre qu'un port USB hôte standard pour alimenter le module par le connecteur USB, vous devez respecter les caractéristiques assignées par le standard USB 2.0:

- **Tension min.:** 4.75 V DC
- **Tension max.:** 5.25 V DC
- **Protection contre les surintensités:** max. 5.0 A

En cas de tension supérieure, le module risque fort d'être détruit. En cas de tension inférieure, le comportement n'est pas déterminé, mais il peut conduire à une corruption du firmware.

Le Yocto-bouton

Le Yocto-bouton a deux fonctions. Premièrement, il permet d'activer la Yocto-balise (voir la Yocto-led ci-dessous). Deuxièmement, si vous branchez un Yocto-module en maintenant ce bouton appuyé, il vous sera possible de reprogrammer son firmware avec une nouvelle version. Notez qu'il existe une méthode plus simple pour mettre à jour le firmware depuis l'interface utilisateur, mais cette méthode-là peut fonctionner même lorsque le firmware chargé sur le module est incomplet ou corrompu.

La Yocto-Led

En temps normal la Yocto-Led sert à indiquer le bon fonctionnement du module: elle émet alors une faible lumière bleue qui varie lentement mimant ainsi une respiration. La Yocto-Led cesse de respirer lorsque le module ne communique plus, par exemple s'il est alimenté par un hub sans connexion avec un ordinateur allumé.

Lorsque vous appuyez sur le Yocto-bouton, la Led passe en mode Yocto-balise: elle se met alors à flasher plus vite et beaucoup plus fort, dans le but de permettre une localisation facile d'un module lorsqu'on en a plusieurs identiques. Il est en effet possible de déclencher la Yocto-balise par logiciel, tout comme il est possible de détecter par logiciel une Yocto-balise allumée.

La Yocto-Led a une troisième fonctionnalité moins plaisante: lorsque ce logiciel interne qui contrôle le module rencontre une erreur fatale, elle se met à flasher SOS en morse[1]. Si cela arrivait débranchez puis rebranchez le module. Si le problème venait à se reproduire vérifiez que le module contient bien la dernière version du firmware, et dans l'affirmative contactez le support Yoctopuce[2].

La sonde de courant

Chaque Yocto-module est capable de mesurer sa propre consommation de courant sur le bus USB. La distribution du courant sur un bus USB étant relativement critique, cette fonctionnalité peut être d'un grand secours. La consommation de courant du module est consultable par logiciel uniquement.

Le numéro de série

Chaque Yocto-module a un numéro de série unique attribué en usine, pour les modules Yocto-SDI12 ce numéro commence par YSDIMK01. Le module peut être piloté par logiciel en utilisant ce numéro de série. Ce numéro de série ne peut pas être changé.

Le nom logique

Le nom logique est similaire au numéro de série, c'est une chaîne de caractères sensée être unique qui permet référencer le module par logiciel. Cependant, contrairement au numéro de série, le nom logique peut être modifié à volonté. L'intérêt est de pouvoir fabriquer plusieurs exemplaires du même projet sans avoir à modifier le logiciel de pilotage. Il suffit de programmer les mêmes noms logiques dans chaque exemplaire. Attention, le comportement d'un projet devient imprévisible s'il contient plusieurs modules avec le même nom logique et que le logiciel de pilotage essaye d'accéder à l'un de ces module à l'aide de son nom logique. A leur sortie d'usine, les modules n'ont pas de nom logique assigné, c'est à vous de le définir.

2.2. Les éléments spécifiques

Le connecteur

Le module Yocto-SDI12 dispose d'un connecteur classique SDI-12 avec trois lignes:

- Une ligne 12V pour alimenter les capteurs SDI-12 si besoin est.
- Une ligne d'entrée/sortie de données, pour communiquer avec les capteurs SDI-12.
- La masse (GND).

Il n'est pas nécessaire de connecter la ligne 12V si le capteur est alimenté par une alimentation externe, cependant le GND du Yocto-SDI12 doit être en commun avec celui du capteur pour assurer la communication avec le capteur.

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

Les LEDs d'activité

Le Yocto-SDI12 dispose de deux LEDs vertes reflétant l'activité du port SDI-12. Une LED TX pour les données envoyées et une LED RX pour les données reçues par le Yocto-SDI12.

2.3. Accessoires optionnels

Les accessoires ci-dessous ne sont pas nécessaires à l'utilisation du module Yocto-SDI12, mais pourraient vous être utiles selon l'utilisation que vous en faites. Il s'agit en général de produits courants que vous pouvez vous procurer chez vos fournisseurs habituels de matériel de bricolage. Pour vous éviter des recherches, ces produits sont en général aussi disponibles sur le shop de Yoctopuce.

Vis et entretoises

Pour fixer le module Yocto-SDI12 à un support, vous pouvez placer des petites vis de 2.5mm avec une tête de 4.5mm au maximum dans les trous prévus ad-hoc. Il est conseillé de les visser dans des entretoises filetées, que vous pourrez fixer sur le support. Vous trouverez plus de détail à ce sujet dans le chapitre concernant le montage et la connectique.

Micro-hub USB

Si vous désirez placer plusieurs modules Yoctopuce dans un espace très restreint, vous pouvez les connecter ensemble à l'aide d'un micro-hub USB. Yoctopuce fabrique des hubs particulièrement petits précisément destinés à cet usage, dont la taille peut être réduite à 20mm par 36mm, et qui se montent en soudant directement les modules au hub via des connecteurs droits ou des câbles nappe. Pour plus de détails, consulter la fiche produit du micro-hub USB.

YoctoHub-Ethernet, YoctoHub-Wireless and YoctoHub-GSM

Vous pouvez ajouter une connectivité réseau à votre Yocto-SDI12 grâce aux hubs YoctoHub-Ethernet, YoctoHub-Wireless et YoctoHub-GSM qui offrent respectivement une connectivité Ethernet, Wifi et GSM. Chacun de ces hubs peut piloter jusqu'à trois modules Yoctopuce et se comporte exactement comme un ordinateur normal qui ferait tourner VirtualHub.

Connecteurs 1.27mm (ou 1.25mm)

Si vous désirez raccorder le module Yocto-SDI12 à un Micro-hub USB ou a un YoctoHub en évitant l'encombrement d'un vrai cable USB, vous pouvez utiliser les 4 pads au pas 1.27mm juste derrière le connecteur USB. Vous avez alors deux possibilités.

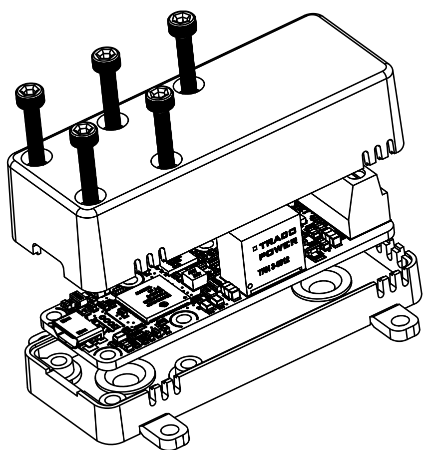
Vous pouvez monter directement le module sur le hub à l'aide d'un jeu de vis et entretoises, et les connecter à l'aide de connecteurs board-to-board au pas 1.27mm. Pour éviter les court-circuits, soudez de préférence le connecteur femelle sur le hub et le connecteur mâle sur le Yocto-SDI12.

Vous pouvez aussi utiliser un petit câble à 4 fils doté de connecteurs au pas 1.27mm (ou 1.25mm, la différence est négligeable pour 4 pins), ce qui vous permet de déporter le module d'une dizaine de centimètres. N'allongez pas trop la distance si vous utilisez ce genre de câble, car il n'est pas blindé et risque donc de provoquer des émissions électromagnétiques indésirables.

Boîtier

Votre Yocto-SDI12 a été conçu pour pouvoir être installé tel quel dans votre projet. Néanmoins Yoctopuce commercialise des boîtiers spécialement conçus pour les modules Yoctopuce. Ces boîtiers sont munis de pattes de fixation amovibles et d'aimants de fixation. Vous trouverez plus d'informations à propos de ces boîtiers sur le site de Yoctopuce[3]. Le boîtier recommandé pour votre Yocto-SDI12 est le modèle YoctoBox-Long-Thick-Black

[3] https://www.yoctopuce.com/FR/products/category/boitiers



Vous pouvez installer votre Yocto-SDI12 dans un boîtier optionnel.

3. Premiers pas

Par design, tous les modules Yoctopuce se pilotent de la même façon, c'est pourquoi les documentations des modules de la gamme sont très semblables. Si vous avez déjà épluché la documentation d'un autre module Yoctopuce, vous pouvez directement sauter à la description de sa configuration.

3.1. Prérequis

Pour pouvoir profiter pleinement de votre module Yocto-SDI12, vous devriez disposer des éléments suivants.

Un ordinateur

Les modules de Yoctopuce sont destinés à être pilotés par un ordinateur (ou éventuellement un microprocesseur embarqué). Vous écrirez vous-même le programme qui pilotera le module selon vos besoins, à l'aide des informations fournies dans ce manuel.

Yoctopuce fourni les librairies logicielles permettant de piloter ses modules pour les systèmes d'exploitation suivants: **Windows, Linux, macOS et Android**. Les modules Yoctopuce ne nécessitent pas l'installation de driver (ou pilote) spécifiques, car ils utilisent le driver HID[1] fourni en standard dans tous les systèmes d'exploitation.

La règle générale concernant les versions de système d'exploitation supportées est la suivante: les outils de développement Yoctopuce sont supportés pour toutes les versions couvertes par le support de l'éditeur du système d'exploitation, y compris la durée du support étendu (*long term support* ou LTS). Yoctopuce attache une attention particulière au support à long terme, et lorsque c'est possible avec un effort raisonnable, nos outils sont construits de sorte à pouvoir être utilisés sur des anciens systèmes même plusieurs années encore après la fin du support étendu par le fabricant.

De plus, les librairies de programmation pour piloter nos modules étant disponibles en code source, il vous est en généralement possible de les recompiler pour fonctionner sur des systèmes d'exploitation encore plus anciens. A ce jour, notre librairie de programmation peut toujours être compilée pour fonctionner sur des systèmes d'exploitation publiés en 2008, tels que Windows XP SP3 ou Linux Debian Squeeze.

Les architectures supportées par les librairies logicielles de Yoctopuce sont les suivantes:

- Windows: Intel 64 bits et 32 bits

[1] Le driver HID est celui qui gère les périphériques tels que la souris, le clavier, etc.

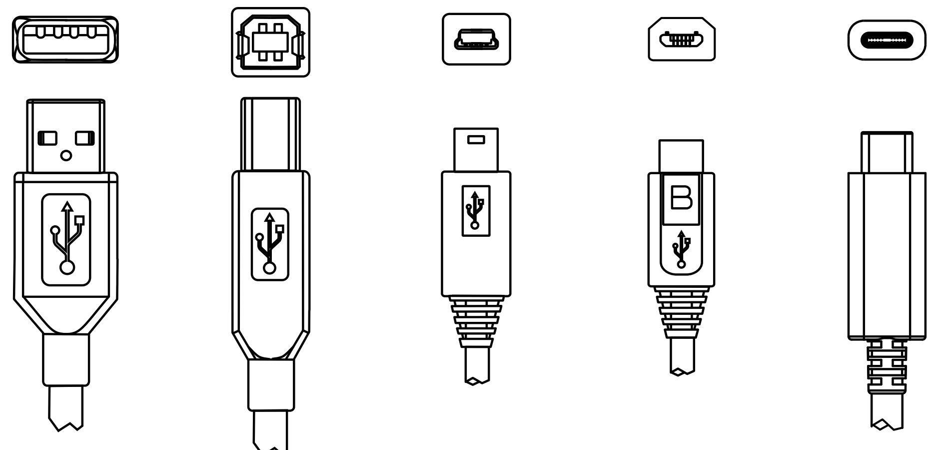
- Linux: Intel 64 bits et 32 bits, ARM 64 bits et 32 bits, y compris Raspberry Pi OS.
- macOS: Intel 64 bits et Apple Silicon (ARM)

Sous Linux, la communication avec nos modules USB requiert impérativement la librairie libusb en version 1.0 ou plus récente, qui est disponible sur toutes les distributions courantes. Les librairies et les outils en ligne de commande devraient pouvoir être facilement recompilés sur n'importe quelle variante d'UNIX (Linux, FreeBSD, ...) datant des quinze dernières années pour laquelle libusb-1.0 est disponible et fonctionnel.

Sous Android, la possibilité de connecter un module USB dépend du fait que la tablette ou le téléphone supporte le *mode USB Host*.

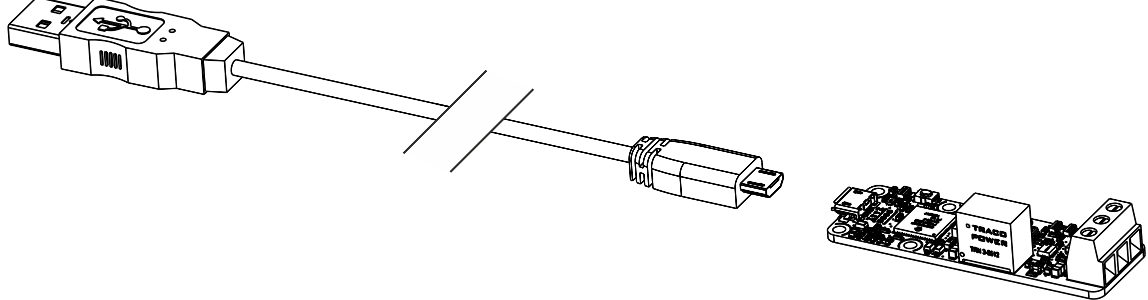
Un cable USB 2.0 de type A-micro B

Il existe plusieurs formes de connecteurs USB. La taille "normale" correspond à celle que vous utilisez probablement pour brancher votre imprimante. La taille "mini" a plus ou moins disparu. La taille "micro" était la plus petite au moment où les modules Yoctopuce sont apparus, et c'est toujours celle que nous utilisons. Depuis quelques années, des connecteurs USB-C sont apparus, mais pour ne pas multiplier le type de connecteurs dans notre gamme de produit, nous sommes pour l'instant resté au standard "micro-B".



Les connecteurs USB 2.0 les plus courants: A, B, Mini B, Micro B et USB-C.

Pour connecter votre module Yocto-SDI12 à un ordinateur, vous avez donc besoin d'un cable USB 2.0 de type A-micro B. Vous trouverez ce cable en vente à des prix très variables selon les sources, sous la dénomination *USB A to micro B Data cable*. Prenez garde à ne pas acheter par mégarde un simple câble de charge, qui ne fournirait que le courant mais sans les fils de données. Le bon câble est disponible sur le shop de Yoctopuce.



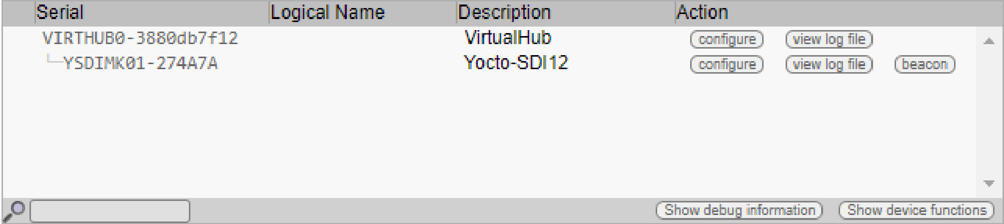
Vous devez raccorder votre module Yocto-SDI12 à l'aide d'un cable USB 2.0 de type A - micro B

Si vous branchez un hub USB entre l'ordinateur et le module Yocto-SDI12, prenez garde à ne pas dépasser les limites de courant imposées par USB, sous peine de faire face des comportements instables non prévisibles. Vous trouverez plus de détails à ce sujet dans le chapitre concernant le montage et la connectique.

3.2. Test de la connectivité USB

Arrivé à ce point, votre Yocto-SDI12 devrait être branché à votre ordinateur, qui devrait l'avoir reconnu. Il est temps de le faire fonctionner.

Rendez-vous sur le site de Yoctopuce et téléchargez le programme *VirtualHub*[2], Il est disponible pour Windows, Linux et macOS. En temps normal le programme VirtualHub sert de couche d'abstraction pour les langages qui ne peuvent pas accéder aux couches matérielles de votre ordinateur. Mais il offre aussi une interface sommaire pour configurer vos modules et tester les fonctions de base, on accède à cette interface à l'aide d'un simple browser web [3]. Lancez VirtualHub en ligne de commande, ouvrez votre browser préféré et tapez l'adresse *http://127.0.0.1:4444*. Vous devriez voir apparaître la liste des modules Yoctopuce raccordés à votre ordinateur.



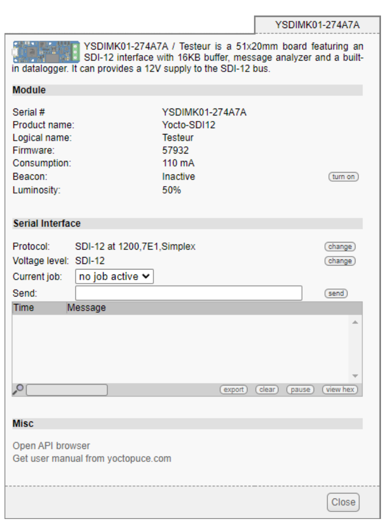
Liste des modules telle qu'elle apparaît dans votre browser.

3.3. Localisation

Il est alors possible de localiser physiquement chacun des modules affichés en cliquant sur le bouton **beacon**, cela a pour effet de mettre la Yocto-Led du module correspondant en mode "balise", elle se met alors à clignoter ce qui permet de la localiser facilement. Cela a aussi pour effet d'afficher une petite pastille bleue à l'écran. Vous obtiendrez le même comportement en appuyant sur le Yocto-bouton d'un module.

3.4. Test du module

La première chose à vérifier est le bon fonctionnement de votre module: cliquez sur le numéro de série correspondant à votre module, et une fenêtre résumant les propriétés de votre Yocto-SDI12.



Propriétés du module Yocto-SDI12.

Cette fenêtre vous permet, entre autres, de jouer avec votre module pour en vérifier son fonctionnement: vous y trouverez un émulateur de terminal simplifié vous permettant de tester les

[2] www.yoctopuce.com/FR/virtualhub.php
[3] L'interface est testée avec Chrome, FireFox, Safari, Edge et IE 11.

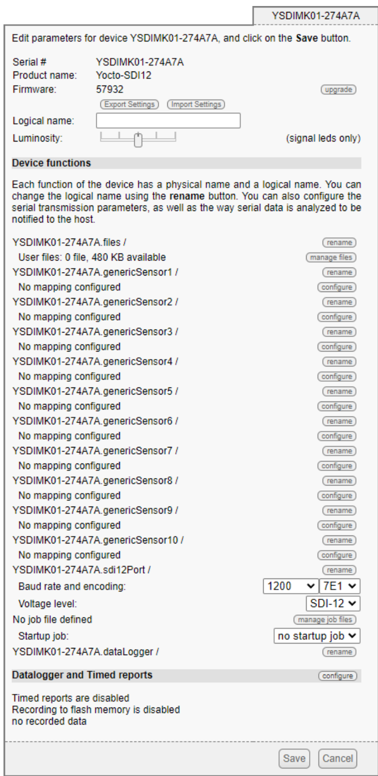
communications de votre module. Pour commencer, si vous avez un capteur branché à votre module, vous pouvez essayer d'envoyer la commande **?!** qui demande au capteur son adresse.

En supposant que le capteur vous a répondu que son adresse est le **3**, vous pouvez lui envoyer par exemple la commande **3I!** pour lui demander qui il est, ou **3M!** pour lui demander d'effectuer une mesure.

Pour plus de détails sur les messages SDI-12 que vous pouvez envoyer, référez-vous au chapitre 5 intitulé *Le port SDI-12*.

3.5. Configuration

Si, dans la liste de modules, vous cliquez sur le bouton **configure** correspondant à votre module, la fenêtre de configuration apparaît.



Configuration du module Yocto-SDI12.

Firmware

Le firmware du module peut être facilement mis à jour à l'aide de l'interface. Les firmwares destinés aux modules Yoctopuce se présentent sous la forme de fichiers .byn et peuvent être téléchargés depuis le site web de Yoctopuce.

Pour mettre à jour un firmware, cliquez simplement sur le bouton **upgrade** de la fenêtre de configuration et suivez les instructions. Si pour une raison ou une autre, la mise à jour venait à échouer, débranchez puis rebranchez le module. Recommencer la procédure devrait résoudre alors le problème. Si le module a été débranché alors qu'il était en cours de reprogrammation, il ne fonctionnera probablement plus et ne sera plus listé dans l'interface. Mais il sera toujours possible de le reprogrammer correctement en utilisant le programme VirtualHub[4] en ligne de commande [5].

[4] www.yoctopuce.com/FR/virtualhub.php
[5] Consultez la documentation de VirtualHub pour plus de détails

Nom logique du module

Le nom logique est un nom choisi par vous, qui vous permettra d'accéder à votre module, de la même manière qu'un nom de fichier vous permet d'accéder à son contenu. Un nom logique doit faire au maximum 19 caractères, les caractères autorisés sont les caractères `A..Z a..z 0..9 _` et `-`. Si vous donnez le même nom logique à deux modules raccordés au même ordinateur, et que vous tentez d'accéder à l'un des modules à l'aide de ce nom logique, le comportement est indéterminé: vous n'avez aucun moyen de savoir lequel des deux va répondre.

Luminosité

Ce paramètre vous permet d'agir sur l'intensité maximale des leds présentes sur le module. Ce qui vous permet, si nécessaire, de le rendre un peu plus discret tout en limitant sa consommation. Notez que ce paramètre agit sur toutes les leds de signalisation du module, y compris la Yocto-Led. Si vous branchez un module et que rien ne s'allume, cela veut peut être dire que sa luminosité a été réglée à zéro.

Nom logique des fonctions

Chaque module Yoctopuce a un numéro de série, et un nom logique. De manière analogue, chaque fonction présente sur chaque module Yoctopuce a un nom matériel et un nom logique, ce dernier pouvant être librement choisi par l'utilisateur. Utiliser des noms logiques pour les fonctions permet une plus grande flexibilité au niveau de la programmation des modules

Configuration du port SDI-12

Cette fenêtre vous permet de configurer le fonctionnement du port SDI-12. La configuration la plus commune consiste en une vitesse de 1200 baud avec 1 bit de start, 7 bits de data en LSB, 1 bit de parité paire et 1 bit de stop. Cependant il peut arriver que des capteurs fonctionnent avec une configuration différente. Dans ce cas, la fenêtre de configuration vous permettra de modifier ces paramètres en conséquence.

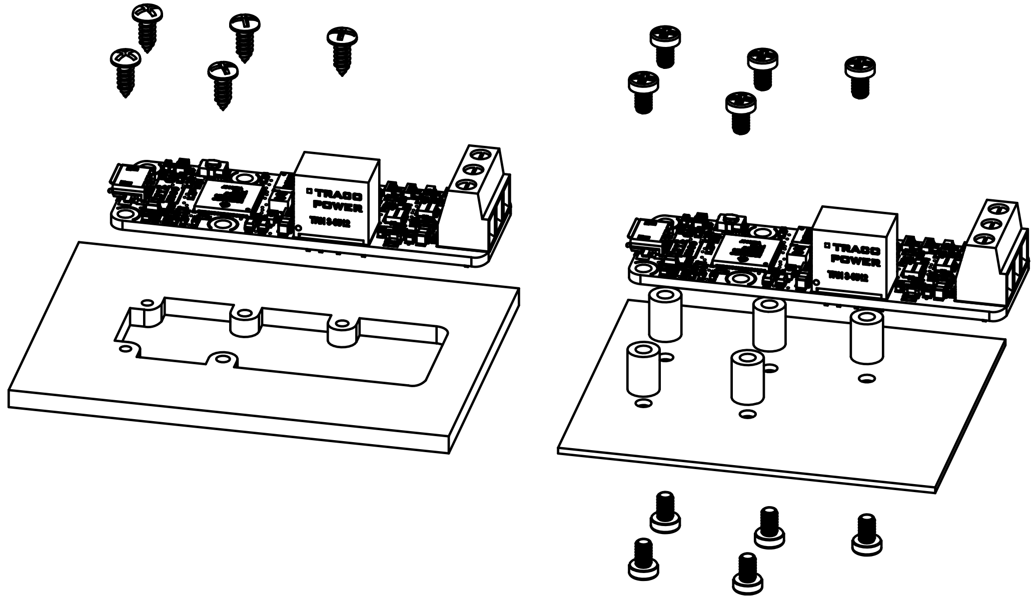
Le Yocto-SDI12 dispose d'une sortie capable d'alimenter les capteurs connectés au bus SDI-12 en 12 V dans la limite de 200mA. L'alimentation 12V peut être allumée ou éteinte à volonté, ce qui peut vous permettre de réinitialiser les capteurs connectés en cas de problème de communication. A noter que la communication ne fonctionnera pas si la sortie 12V n'est pas active, même si les capteurs sont alimentés par une source externe.

4. Montage et connectique

Ce chapitre fournit des explications importantes pour utiliser votre module Yocto-SDI12 en situation réelle. Prenez soin de le lire avant d'aller trop loin dans votre projet si vous voulez éviter les mauvaises surprises.

4.1. Fixation

Pendant la mise au point de votre projet vous pouvez vous contenter de laisser le module se promener au bout de son câble. Veillez simplement à ce qu'il ne soit pas en contact avec quoi que soit de conducteur (comme vos outils). Une fois votre projet pratiquement terminé il faudra penser à faire en sorte que vos modules ne puissent pas se promener à l'intérieur.



Exemples de montage sur un support.

Le module Yocto-SDI12 dispose de trous de montage 2.5mm. Vous pouvez utiliser ces trous pour y passer des vis. Le diamètre de la tête de ces vis ne devra pas dépasser 4.5mm, sous peine d'endommager les circuits du module. Veillez à que la surface inférieure du module ne soit pas en contact avec le support. La méthode recommandée consiste à utiliser des entretoises, mais il en existe d'autres. Rien ne vous empêche de le fixer au pistolet à colle; ça ne sera pas très joli mais ça tiendra.

Si vous comptez visser votre module directement contre une paroi conductrice, un chassis métallique par exemple, intercalez une couche isolante entre les deux. Sinon vous aller à coup sûr provoquer un court-circuit: il y a des pads à nu sous votre module. Du simple ruban adhésif isolant devrait faire l'affaire.

4.2. Contraintes d'alimentation par USB

Bien que USB signifie *Universal Serial BUS*, les périphériques USB ne sont pas organisés physiquement en bus mais en arbre, avec des connections point-à-point. Cela a des conséquences en termes de distribution électrique: en simplifiant, chaque port USB doit alimenter électriquement tous les périphériques qui lui sont directement ou indirectement connectés. Et USB impose des limites.

En théorie, un port USB fournit 100mA, et peut lui fournir (à sa guise) jusqu'à 500mA si le périphérique les réclame explicitement. Dans le cas d'un hub non-alimenté, il a droit à 100mA pour lui-même et doit permettre à chacun de ses 4 ports d'utiliser 100mA au maximum. C'est tout, et c'est pas beaucoup. Cela veut dire en particulier qu'en théorie, brancher deux hub USB non-alimentés en cascade ne marche pas. Pour cascader des hubs USB, il faut utiliser des hubs USB alimentés, qui offriront 500mA sur chaque port.

En pratique, USB n'aurait pas eu le succès qu'il a s'il était si contraignant. Il se trouve que par économie, les fabricants de hubs omettent presque toujours d'implémenter la limitation de courant sur les ports: ils se contentent de connecter l'alimentation de tous les ports directement à l'ordinateur, tout en se déclarant comme *hub alimenté* même lorsqu'ils ne le sont pas (afin de désactiver tous les contrôles de consommation dans le système d'exploitation). C'est assez malpropre, mais dans la mesure où les ports des ordinateurs sont eux en général protégés par une limitation de courant matérielle vers 2000mA, ça ne marche pas trop mal, et cela fait rarement des dégâts.

Ce que vous devez en retenir: si vous branchez des modules Yoctopuce via un ou des hubs non alimentés, vous n'aurez aucun garde-fou et dépendrez entièrement du soin qu'aura mis le fabricant de votre ordinateur pour fournir un maximum de courant sur les ports USB et signaler les excès avant qu'ils ne conduisent à des pannes ou des dégâts matériels. Si les modules sont sous-alimentés, ils pourraient avoir un comportement bizarre et produire des pannes ou des bugs peu reproductibles. Si vous voulez éviter tout risque, ne cascadez pas les hubs non-alimentés, et ne branchez pas de périphérique consommant plus de 100mA derrière un hub non-alimenté.

Pour vous faciliter le contrôle et la planification de la consommation totale de votre projet, tous les modules Yoctopuce sont équipés d'une sonde de courant qui indique (à 5mA près) la consommation du module sur le bus USB.

Notez enfin que le câble USB lui-même peut aussi représenter une cause de problème d'alimentation, en particulier si les fils sont trop fins ou si le câble est trop long [1]. Les bons câbles utilisent en général des fils AWG 26 ou AWG 28 pour les fils de données et des fils AWG 24 pour les fils d'alimentation.

4.3. Compatibilité électromagnétique (EMI)

Les choix de connectique pour intégrer le Yocto-SDI12 ont naturellement une incidence sur les émissions électromagnétiques du système, et donc sur la conformité avec les normes concernées.

Les mesures de référence que nous effectuons pour valider la conformité avec la norme IEC CISPR 11 sont faites sans aucun boîtier, mais en raccordant les modules par un câble USB blindé, conforme à la spécification USB 2.0: le blindage du câble est relié au blindage des deux connecteurs, et la résistance totale entre le blindage des deux connecteurs est inférieure 0.6Ω. Le câble utilisé fait 3m, de sorte à exposer un segment d'un mètre horizontal, un segment d'un mètre vertical et de garder le dernier mètre le plus proche de l'ordinateur hôte à l'intérieur d'un bloc de ferrite.

[1] www.yoctopuce.com/FR/article/cables-usb-la-taille-compte

Si vous utilisez un câble non blindé ou incorrectement blindé, votre système fonctionnera sans problème mais vous risquez de n'être pas conforme à la norme. Dans le cadre de systèmes composés de plusieurs modules raccordés par des câbles au pas 1.27mm, ou de capteurs déportés, vous pourrez en général récupérer la conformité avec la norme d'émission en utilisant un boîtier métallique offrant une enveloppe de blindage externe.

Toujours par rapport aux normes de compatibilité électromagnétique, la longueur maximale supportée du câble USB est de 3m. En plus de pouvoir causer des problèmes de chute de tension, l'utilisation de câbles plus long aurait des incidences sur les tests d'immunité électromagnétique à effectuer pour respecter les normes.

5. Le port SDI-12

Ce chapitre rappelle le fonctionnement d'un bus SDI-12 et décrit les abstractions utilisées pour implémenter différentes interfaces de contrôle SDI-12 dans l'API Yoctopuce.

5.1. Fonctionnement d'un bus SDI-12

Un bus SDI-12 est constitué de trois lignes:

- La masse;
- L'alimentation, généralement 12 V;
- La ligne qui transmet le signal de données.

Une interface SDI-12 communique par échange de caractères ASCII sur la ligne de données entre un maître et un capteur ou plusieurs. Pour démarrer une communication, le maître envoie une commande pause pour réveiller les capteurs qui sont sur la ligne de données. Une pause est un espacement continue sur la ligne de données pendant au moins 12 millisecondes. Le maître envoie ensuite une commande à un capteur spécifique. Le capteur renvoie à son tour la réponse appropriée. Le premier caractère de chaque commande est une adresse de capteur unique qui précise avec quel capteur le maître veut communiquer. Les autres capteurs connectés sur la ligne ignorent la commande parce qu'elle ne leur est pas destinée et reviennent en mode veille.

Le bus SDI-12 est généralement piloté par un maître qui commande un ou plusieurs capteurs chacun son tour. Le Yocto-SDI12 est conçu pour assumer le rôle de maître.

5.2. Baud Rate et Bytes Frame Format

Le baud rate du bus SDI-12 est de 1200, pour une communication normale, et est composé de 1 bit de start, suivi de 7 bits de données, puis 1 bit de parité paire et pour finir 1 bit de stop.

5.3. Communication entre un maître et un capteur

La majorité des communications SDI-12 entre un maître et un capteur se déroule dans cet ordre:

- Le maître réveille tous les capteurs du port SDI-12 avec l'envoi d'un signal *break*.
- Le maître transmet une commande à un capteur spécifique adressé.
- Le capteur adressé répond dans les 15 millisecondes en envoyant sa réponse.

Le protocol SDI-12 impose des temporisations très précises à respecter pour avoir une communication entre le capteur et le maître. Pour vous simplifier la vie, tous les timings de

communication sont automatiquement gérés par le Yocto-SDI12. Chaque début de communication commence par un signal break d'au moins 12 millisecondes, pour réveiller le capteur, suivi d'une attente de 8,33 millisecondes puis la commande est envoyée. Le capteur a jusqu'à 15 millisecondes pour répondre avant que le Yocto-SDI12 ne fasse automatiquement une relance pour avoir une réponse du capteur. Le Yocto-SDI12 effectue trois relances avant de déclarer que le capteur n'est pas joignable.

5.4. Caractères autorisés

Tous les caractères transmis sur le bus SDI-12 doivent être des caractères ASCII imprimables. Il y a trois exceptions:

- Toutes les réponses d'un capteur SDI-12 se terminent avec un retour chariot et un caractère de saut de ligne, représenté par <CR> (code ASCII 13); <LF> (code ASCII 10) dans ce document;
- Pour certaines commandes, il est possible d'avoir un CRC avec la réponse du capteur, ces dernières peuvent contenir des caractères ASCII non imprimables;
- Le contenu des paquets de données renvoyé par la commande de high volume binary.

5.5. Adresse

Le premier caractère envoyé de chaque commande est l'adresse du capteur. De la même manière, le premier caractère reçu du capteur sera aussi son adresse. Les adresses valides sont de '0' à '9', 'a' à 'z' et 'A' à 'Z' (case sensitive), l'adresse par défaut d'un capteur est généralement '0'.

5.6. Commandes SDI-12

Les commandes SDI-12 finissent toujours avec le caractère '!' pour indiquer la fin de la commande à envoyer. La réponse du capteur, elle, finit toujours par un retour chariot et un caractère de saut de ligne. Le protocole SDI-12 décrit plusieurs commandes de base que tous les capteurs sont sensés implémenter, listées ci-dessous avec leur description et le format d'envoi (en gras) et de réception ainsi qu'un exemple de la commande avec sa réponse.

Address Query Command (?!)

Cette commande sert à trouver l'adresse du capteur connecté. Elle est très pratique pour tester un nouveau capteur. Attention cette fonction fonctionne uniquement si un seul capteur est connecté sur le bus SDI-12. Pour utiliser cette commande, le module envoie les caractères '?' et '!'. Le capteur répond avec son adresse.

	Commande	Réponse
Format général:	**?!**	a<CR><LF>
Exemple pratique:	**?!**	1<CR><LF>

Send Identification Command (aI!)

Cette commande sert à avoir toutes les informations du capteur interrogé. Le module envoie l'adresse du capteur suivi des caractères 'I' et '!'. La réponse du capteur est la suivante:

- a - l'adresse du capteur
- ll - la version SDI-12 du capteur
- cccccccc - les informations de la compagnie qui produit le capteur
- vvv - la version du capteur
- xxx...xxx - un espace optionnel de 13 caractères

	Commande	Réponse
Format général:	**aI!**	allccccccccmmmmmmvvvxxx...xxx<CR><LF>

Exemple pratique:	**1I!**	113Delta-T WET150v03 D0002299<CR><LF>

Acknowledge Active Command (a!)

Cette commande sert à confirmer que le capteur est bien connecté et réveillé. Le module envoie l'adresse du capteur suivi du caractère '!'. Le capteur répond avec son adresse et avec un retour chariot et un caractère de saut de ligne.

	Commande	Réponse
Format général:	**a!**	a<CR><LF>
Exemple pratique:	**1!**	1<CR><LF>

Change Address Command (aAb!)

Cette commande sert à change l'adresse d'un capteur. Le module envoie l'adresse actuelle du capteur suivi du caractère 'A' et de la nouvelle adresse souhaitée et finit la commande avec le caractère '!'. Le capteur répond avec sa nouvelle adresse.

	Commande	Réponse
Format général:	**aAb!**	b<CR><LF>
Exemple pratique:	**0A1!**	1<CR><LF>

Start Measurement Command (aM!)

Cette commande sert demander à un capteur de faire une mesure, aucune autre mesure ne peut être effectuée avant d'avoir le résultat de cette mesure. Le module envoie l'adresse du capteur suivi des caractères 'M' et '!'. Le capteur répond avec son adresse suivie du temps d'attente, en secondes, pour le résultat de la mesure, sur trois caractères, et du nombre de valeurs à mesurer sur un caractère.

	Commande	Réponse
Format général:	**aM!**	atttn<CR><LF>
Exemple pratique:	**1M!**	10035<CR><LF>

Si le capteur envoie 000 sur le temps d'attente alors les valeurs sont immédiatement prête a être lue avec la commande **D0**. Sinon il faudra attendre le temps indiqué pour lire les valeurs du capteur avec la commande **D0**. Il peut aussi arriver que les valeurs soient prêtes avant le temps donné. Dans ce cas-là, le capteur envoie son adresse sur le bus SDI-12 pour notifier que la lecture prête. Cette commande est aussi disponible avec un CRC dans le résultat des valeurs du capteur. Pour cela, il suffit juste d'envoyer la commande **aMC!**, la réponse du capteur sera la même que pour la commande **aM!**.

Start Concurrent Measurement Command (aC!)

Cette commande sert à demander à un capteur de faire une mesure pendant qu'un autre capteur effectue une mesure. Attention, pour qu'il n'y ait pas de conflit, il faut que toutes les mesures simultanées soient déclenchées par une commande **C**. Le module envoie l'adresse du capteur suivi des caractères 'C' et '!'. Comme pour la commande **M**, le capteur ne renvoie pas les valeurs de la mesure mais son adresse suivie du temps d'attente, en secondes, pour le résultat de la mesure, sur trois caractères, et du nombre de valeurs à mesurer sur deux caractères.

	Commande	Réponse
Format général:	**aC!**	atttnn<CR><LF>
Exemple pratique:	**1C!**	100305<CR><LF>

Si le capteur envoie 000 sur le temps d'attente alors les valeurs sont immédiatement prête à être lue avec la commande **D0**. Sinon il faudra attendre le temps reçut pour lire les valeurs du capteur avec la commande **D0**. Cette commande est aussi disponible avec un CRC dans le résultat des valeurs du capteur. Pour cela il suffit juste d'envoyer la commande **aCC!**, la réponse du capteur sera la même que pour la commande **aC!**.

Additional Measurement Commands (aM1!, aM2!, ...)

Les commandes **M** supplémentaires permettent de demander différents types de mesure au capteur. Chaque capteur a des mesures supplémentaires, pour obtenir ses types de mesure il faut étudier la documentation du capteur. Si le capteur n'a pas de mesure complémentaire il renvoie simplement a0000<CR><LF>. Sinon il répond comme une commande **M**.

Additional Concurrent Measurement Commands (aC1!, aC2!, ...)

Les commandes **C** supplémentaires permettent de demander différents types de mesure au capteur. Chaque capteur a des mesures supplémentaires, pour obtenir ses types de mesure il faut étudier la documentation du capteur. Si le capteur n'a pas de mesure complémentaire il renvoie simplement a00000<CR><LF>. Sinon il répond comme une commande **C**.

Send Data Command (aD0!, aD1!, ...)

Cette commande sert à lire les valeurs des mesures effectuées par le capteur. La commande **D0!** est à appeler après une commande **M, MC, C ou CC**. Pour les longs résultats, il est possible que toutes les valeurs ne soient pas renvoyées avec la commande **D0!**. Il faut donc appeler la commande **D1!** et ainsi de suite jusqu'à avoir toutes les valeurs mesurées. Quand il n'y a plus de valeurs à lire le capteur répond avec uniquement son adresse.

	Commande	Réponse
Format général:	**aD0! (aD1!...aD9!)**	a<values><CR><LF>
Exemple pratique:	**1D0!**	1+0.0+23.9+2.0+0+0<CR><LF>

Les valeurs envoyées par le capteur sont séparées les unes des autres par leur polarité (+ ou -), le nombre maximum de chiffre pour une valeur est de 7 et le minimum est de 1, le nombre maximum de caractères dans une valeur est de 9 en comptant le signe de la polarité les 7 chiffres et la virgule. Cependant, le Yocto-SDI12 attendra automatiquement le résultat des commandes **M** et **V** et enverra automatiquement les commandes **D0!** à **D9!** jusqu'à avoir lu toutes les valeurs du capteur. Le Yocto-SDI12 affichera toute les valeurs reçues par le capteur, puis il affichera une synthèse des réponses du capteur sous la forme suivante:

	Réponse
Format général:	a:val0,val1,val2,val3
Exemple pratique:	1:0.0,23.9,2.0,0

Cette mise en forme facilite la séparation des valeurs reçues.

Read Data (aD!)

La commande **aD!** est une commande spéciale implémentée par Yoctopuce pour lire toutes les valeurs automatiquement et afficher avec la mise en forme suivante:

	Commande	Réponse
Format général:	**aD!**	a:val0,val1,val2,val3
Exemple pratique:	**1D!**	1:0.0,23.9,2.0,0

Cette commande est à utiliser après avoir envoyé une commande qui demande une mesure au capteur (**M,C,V**).

Continuous Measurement Command (aR0!, aR1!, ...)

Cette commande demande à un capteur de retourner une mesure disponible en continu. Les mesures disponibles en continu sont en général des mesures auxiliaires, telles que la tension mesurée sur le bus SDI-12, etc. Le capteur répond avec des valeurs formatées comme pour la commande *Read Data*. Comme pour celle-ci, le Yocto-SDI12 produira en plus une réponse au format séparé par des virgules, plus facile à analyser.

	Commande	Réponse
Format général:	**aR0! (aR1!...aR9!)**	a<values><CR><LF>
Exemple pratique:	**1R0!**	1+12.0<CR><LF>

5.7. Commande de métadonnée

Les commandes de métadonnées ont été introduites avec la version 1.4 de SDI-12. Ces commandes permettent d'obtenir des informations sur les valeurs qui seront retournées par les commandes de mesure. Le capteur doit être à la version 1.4 ou supérieure pour que les commandes de métadonnées fonctionnent. Pour connaître la version du capteur, veuillez consulter la commande *Send Identification Command (aI!)*.

Identify Measurement Commands (aIx!)

Les commandes de d'identification des mesures servent à connaître les informations qui seront retournées par une commande de mesure donnée. Ces commandes d'identification sont composées de l'adresse du capteur suivie de la lettre 'I' et de la commande de mesure souhaitée. La réponse reçue est identique à celle reçue de la commande envoyée sans la lettre 'I'.

Commande	Réponse
aIM! (aIM1!...aIM9!)	atttn<CR><LF>
aIMC! (aIMC1!...aIMC9!)	atttn<CR><LF>
aIV!	atttnn<CR><LF>
aIC! (aIC1!...aIC9!)	atttnn<CR><LF>
aICC! (aICC1!...aICC9!)	atttnn<CR><LF>

	Commande	Réponse
Format général:	**aIM!**	atttn<CR><LF>
Exemple pratique:	**1IM!**	10015<CR><LF>

Identify Measurement Parameter Commands (aIx_xxx!)

Les commandes d'identification des paramètres mesurés permettent de connaître la nature exacte des valeurs mesurées qui seront retournées par une commande. Pour les construire, il suffit d'ajouter la lettre 'I' après l'adresse, suivie de la commande de mesure souhaitée, puis d'un tiret bas (souligné) et du numéro de la valeur souhaitée. Par exemple, pour obtenir les informations sur la deuxième valeur reçue d'une commande **aM!**, il suffit d'appeler la commande **aIM_002!**.

Commande	Réponse
aIM_001!...aIM_009!	a,field1,field2;<CR><LF>
aIM1_001!...aIM1_009!	a,field1,field2;<CR><LF>
...	a,field1,field2;<CR><LF>
aIM9_001!...aIM9_009!	a,field1,field2;<CR><LF>
aIMC_001!...aIMC_009!	a,field1,field2;<CR><LF>
aIMC1_001!...aIMC1_009!	a,field1,field2;<CR><LF>
...	a,field1,field2;<CR><LF>
aIMC9_001!..aIMC9_009!	a,field1,field2;<CR><LF>
aIV_001!...aIV_009!	a,field1,field2;<CR><LF>
aIC1_001!...aIC1_099!	a,field1,field2;<CR><LF>
...	a,field1,field2;<CR><LF>
aIC9_001!...aIC9_099!	a,field1,field2;<CR><LF>
aICC_001!...aICC_099!	a,field1,field2;<CR><LF>
aICC1_001!...aICC1_099!	a,field1,field2;<CR><LF>
...	a,field1,field2;<CR><LF>
aICC9_001!...aICC9_099!	a,field1,field2;<CR><LF>
aIR0_001!...aIR0_099!	a,field1,field2;<CR><LF>
...	a,field1,field2;<CR><LF>
aIR9_001!...aIR9_099!	a,field1,field2;<CR><LF>
aIRC0_001!...aIRC0_099!	a,field1,field2;<CR><LF>

...	a,field1,field2;<CR><LF>
aIRC9_001!...aIRC9_099!	a,field1,field2;<CR><LF>

Les méta-données reçues par le capteur en réponse aux commandes d'information dépendent du capteur lui-même.

	Commande	Réponse
Format général:	**aIM_001!**	a,field1,field2;<CR><LF>
Exemple pratique:	**1IM_001!**	1,MV,%,Soil Moisture;<CR><LF>

5.8. Utilisation à l'aide des fonctions de l'API

L'API Yoctopuce offre plusieurs niveaux d'abstraction pour piloter un capteur communiquant en SDI-12. La manière la plus simple de communiquer avec les capteurs connectés sont d'utiliser les fonctions suivantes.

discoverSingleSensor

La fonction *discoverSingleSensor* sert à trouver l'adresse du capteur connecté sur le Yocto-SDI12, cette fonction ne fonctionne que si un seul capteur est connecté, dans le cas où plusieurs capteurs sont connectés, utiliser la fonction *discoverAllSensors*. La fonction ne prend pas de paramètre d'entrée et renvoie un objet YSdi12SensorInfo avec toutes les informations du capteur. Si aucun capteur n'est connecté, le module renvoie un objet YSdi12SensorInfo avec le texte "Sensor Not Found".

discoverAllSensors

La fonction *discoverAllSensors* sert à trouver toutes les adresses des capteurs connectés sur le Yocto-SDI12. La fonction ne prend pas de paramètre d'entrée et renvoie une liste d'objet YSdi12SensorInfo avec toutes les informations des capteurs. Si aucun capteur n'est connecté, le module renvois un objet YSdi12SensorInfo avec le texte "Sensor Not Found".

getSensorInformation

La fonction *getSensorInformation* sert à avoir toutes les informations d'un capteur. La fonction *getSensorInformation* prend en paramètre l'adresse du capteur, en string, et renvoie un objet YSdi12SensorInfo avec les informations du capteur.

Si le capteur prend en charge les métadonnées introduites avec la version 1.4 du SDI-12, le Yocto-SDI12 interroge automatiquement le capteur pour lire les métadonnées disponibles pour les commandes **M**. Les métadonnées sont géneralement composé des informations sur les valeurs à lire sur le capteur tel que l'unité, le symbole ou encore la descriptions des valeurs luent. Les métadonnées sont renvoyé dans l'objet YSdi12SensorInfo.

readSensor

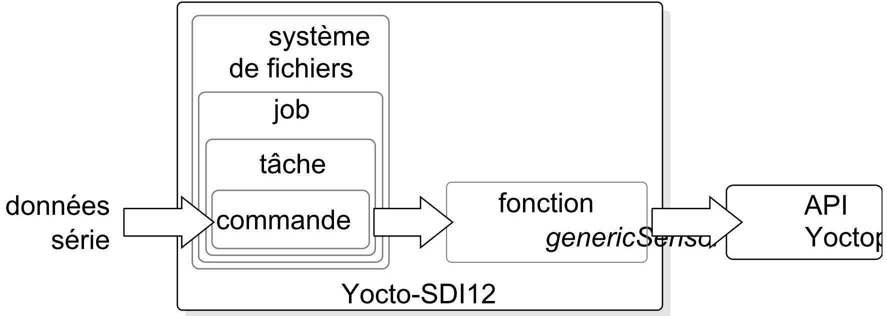
La fonction *readSensor* sert à envoyer une commande et lire la réponse d'un capteur. La fonction *readSensor* prend en paramètre l'adresse du capteur, en string, la commande a envoyé au capteur, en string sans le caractère '!', et le temps maximum à attendre avant la réponse du capteur, en millisecondes. La fonction renvoie un string. Cette fonction est la plus optimale pour lire les valeurs d'un capteur SDI-12.

changeAddress

La fonction *changeAddress* sert à changer l'adresse d'un capteur. Cette fonction prend en premier paramètre l'adresse actuelle du capteur, en string, en deuxième paramètre la nouvelle adresse, en string, et renvoie un objet YSdi12SensorInfo avec les informations du capteur avec sa nouvelle adresse.

6. Mesures automatiques

En plus d'offrir un moyen d'effectuer des communications SDI-12 à bas niveau, le Yocto-SDI12 est capable de travailler à un niveau d'abstraction supérieur. Il peut interroger de manière autonome un capteur SDI-12, et présenter les valeurs lues comme des mesures, à la manière de tous les capteurs Yoctopuce. Cela inclut la possibilité d'enregistrer les mesures sur la mémoire flash interne. Potentiellement, cela permet de transformer n'importe quel capteur doté d'une communication SDI-12 en un capteur Yoctopuce natif avec tous les avantages que cela présente en terme de facilité d'intégration logicielle.



6.1. Les jobs de communication

Le Yocto-SDI12 dispose d'un système de fichiers sur lequel peuvent être stockés des jobs, qui sont en fait de simples fichiers texte au format JSON. Un job décrit des actions d'écriture et de lecture à effectuer sur le port SDI-12.

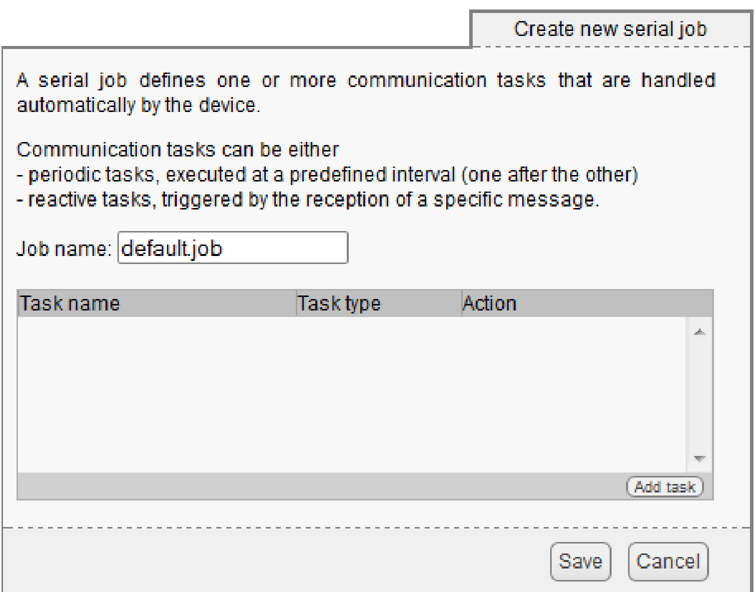
Structure d'un job

Un job est essentiellement un ensemble de tâches qui sont indépendantes les unes des autres. Chaque tâche peut envoyer des données sur le port SDI-12 et/ou réagir à l'arrivée de données du port SDI-12.

Définition et gestion des jobs

Un job se définit à l'aide de VirtualHub, dans la fenêtre configuration du Yocto-SDI12: cliquez simplement sur le bouton **manage files** et une fenêtre contenant la liste des jobs définis apparaîtra.

Cette fenêtre permet de choisir quel job exécuter, d'éditer ou de supprimer des jobs. Elle permet aussi de définir un nouveau job, soit à l'aide d'une interface, soit en l'uploadant directement sur le système de fichiers du module. Pour créer un nouveau job, il suffit donc de cliquer sur le bouton **define a new job** ce qui aura pour effet d'ouvrir la fenêtre de création d'un job.



Fenêtre de création d'un job

Un job n'étant qu'un ensemble de tâches, cette fenêtre ne permet que de donner un nom au job et de gérer les tâches contenues dans le job.

Création d'un job par software
Bien qu'il n'y ait pas d'API explicite pour définir un job par software, un job n'est en fin de compte qu'un fichier texte placé sur le système de fichiers du Yocto-SDI12. Pour configurer Yocto-SDI12 par software, il suffit donc d'uploader le bon fichier sur le Yocto-SDI12 à l'aide de la classe *YFiles* et de programmer son exécution à l'aide des fonctions *selectJob()* ou *set_startupJob()* de la classe *YSdi12Port*. Le moyen le plus simple de créer un fichier job sans risque de faire d'erreur consiste à utiliser VirtualHub pour configurer le job voulu sur un module Yocto-SDI12, et ensuite à downloader le fichier correspondant.

6.2. Les tâches

Chaque tâche est une simple liste de commandes à exécuter séquentiellement: envoyer des données sur le port SDI-12, attendre, lire des données, etc. Il existe essentiellement deux types de tâches : les tâches périodiques et les tâches réactives.

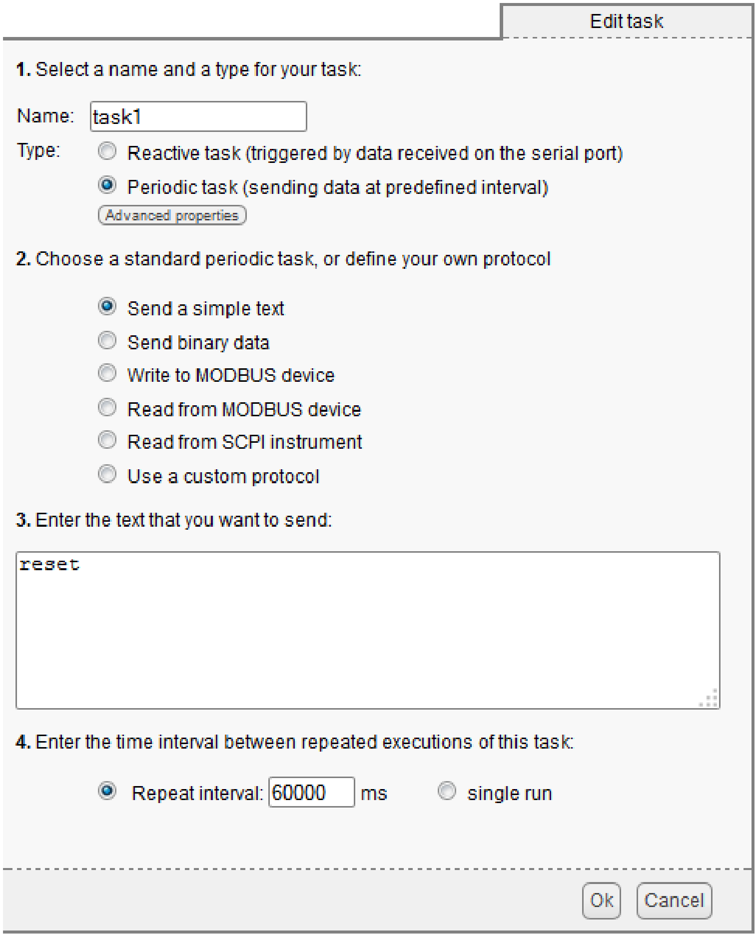
Les tâches périodiques

Une tâche périodique est une tâche qui est exécutée à intervalle régulier, à l'initiative du Yocto-SDI12. Elles sont généralement utilisées pour envoyer des ordres aux capteurs connectés au Yocto-SDI12. Ici encore, VirtualHub permet de définir simplement un certain nombre de tâches usuelles:

- Envoyer une commande
- Lire les valeurs reçues
- Envoyer une commande et lire la réponse.

Il est aussi possible de définir une tâche manuellement, commande par commande, ou de commencer par utiliser une tâche prédéfinie ci-dessus, puis de l'éditer ultérieurement pour ajouter des commandes.

Les données lues lors d'une tâche périodique peuvent être affectées aux fonctions *genericSensor* du Yocto-SDI12.

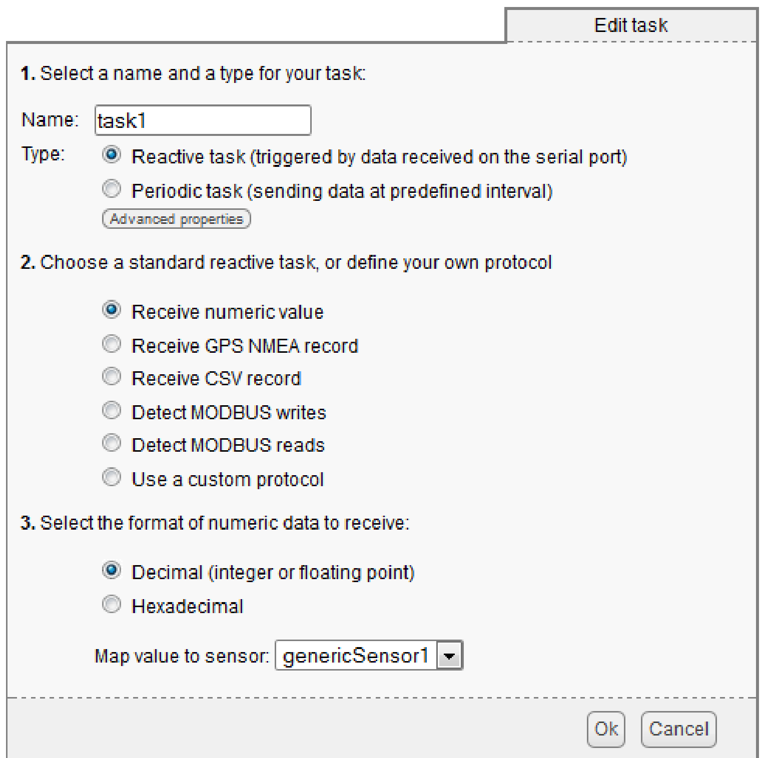


Interface de définition des tâches périodiques

Bien que les tâches périodiques soient conçues pour être exécutées à intervalle régulier, il est possible de définir une tâche "périodique" qui ne sera exécutée qu'une seule fois. L'exécution des tâches périodiques se faisant dans l'ordre de leur définition, il est ainsi de possible définir un job contenant une première tâche, non répétitive, servant à configurer l'instrument et une seconde, répétitive, servant à l'interroger en boucle.

Les tâches réactives

Les tâches réactives se déclenchent lorsqu'un pattern prédéfini à l'aide de la commande *expect* est reconnu. Elles permettent ainsi de réagir de manière asynchrone. Dans le cadre de communications SDI-12, qui sont par nature synchrones et très protocolaires, c'est d'un intérêt limité, sauf pour réagir aux erreurs de communication.



Interface de définition des tâches réactives, ici un exemple qui fait un power cycle du port an cas d'erreur

Il est possible de mixer tâches périodiques et tâches réactives dans un même job, mais il conviendra d'être particulièrement attentif à leurs conditions de déclenchement afin d'éviter qu'elles ne se perturbent les unes les autres. Le Yocto-SDI12 attend toujours qu'une tâche périodique se termine avant de lancer la suivante, mais par contre les tâches réactives peuvent être déclenchées à tout moment, même parallèlement à une tâche périodique.

6.3. Les commandes

Les commandes qui peuvent être utilisées dans une tâche (périodique ou réactive) des modules Yoctopuce gérant une transmission SDI-12 sont les suivantes:

WRITELINE

La commande *writeline* envoie une commande texte, en string, sur le bus SDI-12 suivie d'un retour chariot et un caractère de saut de ligne.

SENDBREAK

La commande *sendbreak* envoie un break sur le bus SDI-12. Cette commande est utilisée pour réveiller le capteur quand il est en mode repos pour pouvoir lui envoyer des commandes. L'argument pour un break standard est à 0 (durée du break = 1 caractère), pour un temps spécifique en millisecondes l'argument doit être entre 1 et 100.

EXPECT

La commande *expect* attend que des données correspondant à un certain pattern apparaissent sur la ligne SDI-12, elle prend en argument une chaîne de caractères. Certaines expressions régulières sont supportées:

- **.** (point) correspond à n'importe quel caractère
- **[]** définit une union, par exemple **[123a-z]** correspond à n'importe quel caractère parmis **1**...**3**,**a**...**z**
- **[^]** définit une exclusion, par exemple **[^,]** correspond à n'importe quel caractère sauf la virgule

- ***** permet de répéter la dernière correspondance zéro, une ou plusieurs fois: **.*** correspond à tous les caractères jusqu'à la fin de la ligne
- **+** permet de répéter la dernière correspondance une ou plusieurs fois. Par exemple, **[0-9]+** correspond à un chiffre ou plus

Des expressions spéciales permettent d'effectuer des décodages et d'affecter la valeur lue à l'un des *genericSensor* du module ou à une variable qui pourra être réutilisée par suite.

- **($x:INT)** permet de reconnaître une valeur entière (en base 10) qui sera affectée à la fonction *genericSensorX*. Par exemple, **{$3:INT}** permet de reconnaître un nombre entier et l'affecter à la fonction *genericSensor3* alors que **{$v:INT}** permet de reconnaître un nombre entier et l'affecter à la variable *v*.
- **($x:FLOAT)** permet de reconnaître une valeur décimale (nombre à virgule), qui sera affectée à la fonction *genericSensorX*. La notation scientifique (par ex. **1.25e-1**) est reconnue.
- **($x:DDM)** permet de reconnaître une valeur décimale en degrés-minutes-décimales telle qu'utilisée dans le standard NMEA.
- **($x:BYTE)** permet de reconnaître une valeur entière entre 0 et 255 codée en hexadécimal (comme c'est le cas pour les protocoles en mode binaire). Si la valeur est dans la plage -128...127, on utilisera **($x:SBYTE)** à la place (*signed byte*). La valeur décodée sera affectée à la fonction *genericSensorX*
- **($x:WORD)** ou **($x:SWORD)** permet de la même manière de décoder une valeur en hexadécimal sur 16 bits, respectivement non signée ou signée, qui sera affectée à la fonction *genericSensorX*. On suppose alors que les octets sont dans l'ordre d'écriture usuel, soit l'octet de poids fort en premier (*big-endian*), comme par exemple 0104 pour représenter la valeur 260.
- **($x:WORDL)** ou **($x:SWORDL)** ont le même effet que les deux précédentes, mais supposent que les octets sont d'en l'ordre *little-endian*, c'est-à-dire l'octet de poids faible en premier (par exemple 0401 pour représenter la valeur 260).
- **($x:DWORD)** ou **($x:SDWORD)** permettent de la même manière de décoder un nombre sur 32 bit en *big-endian* (non-signé ou signé).
- **($x:DWORDL)** ou **($x:SDWORDL)** permettent de la même manière de décoder un nombre sur 32 bit en *little-endian* (non-signé ou signé).
- **($x:DWORDX)** ou **($x:SDWORDX)** permettent de la même manière de décoder un nombre sur 32 bit en *mixed-endian*, soit deux mots de 16 bits chacun représenté en *big-endian*, mais le mot de poids faible en premier et celui de poids fort ensuite.
- **($x:HEX)** permet de reconnaître une valeur en hexadécimal de longueur indéfinie (1 à 4 octets), qui sera affectée à la fonction *genericSensorX*.
- **($x:FLOAT16B)** et **($x:FLOAT16L)** permettent de décoder un nombre flottant codé en hexadécimal selon le standard IEEE 754 sur 16 bits, respectivement avec les octets ordonnés en *big-endian* ou *little-endian*
- **($x:FLOAT16D)** permet de décoder un nombre flottant codé en hexadécimal sur deux octets, avec le premier octet qui contient la mantisse et le deuxième octet qui contient l'exposant décimal signé.
- **($x:FLOAT32B)** et **($x:FLOAT32L)** permettent de décoder un nombre flottant codé en hexadécimal selon le standard IEEE 754 sur 32 bits, respectivement avec les octets ordonnés en *big-endian* ou *little-endian*
- **($x:FLOAT32X)** permet de décoder un nombre flottant codé en hexadécimal selon le standard IEEE 754 sur 32 bits, avec les octets ordonnés en *mixed-endian*, soit deux mots de 16 bits chacun représenté en *big-endian*, mais le mot de poids faible en premier et celui de poids fort ensuite.

La représentation des nombres flottants étant limitée à 3 décimales dans les modules Yoctopuce, il est possible de convertir l'ordre de grandeur des nombres flottants lus par les expressions FLOAT, FLOAT16 et FLOAT32 en les préfixant d'un *M* pour retourner des millièmes, un *U* pour les millionièmes (U comme *micro*) et d'un *N* pour les milliardièmes (N comme *nano*). Ainsi, si l'on reconnait la valeur **1.3e-6** avec l'expression **($1:UFLOAT)**, la valeur affectée au *genericSensor1* sera 1.3.

COMPUTE

La commande *compute* permet de faire des calculs intermédiaires. Par exemple, le code suivant reconnaît un entier et le place dans une variable $t, puis utilise *compute* avec cette variable pour faire une conversion °C/°F et place le résultat dans le GenericSensor n°1.

```
expect  ($t:WORD)
compute $1 = 32 + ($t * 9) / 5
```

Vous pouvez utiliser des expression arithmétiques assez sophistiquées. Tous les opérateurs mathématiques usuels sont disponibles, avec l'ordre de précédence suivant:

<table>
<tr><td><code>**</code></td><td>met à la puissance</td></tr>
<tr><td><code>~ + - not</code></td><td>complément, plus/moins unaire, non logique</td></tr>
<tr><td><code>* / % //</code></td><td>multiplie, divise, modulo et division entière</td></tr>
<tr><td><code>+ -</code></td><td>ajoute, soustrait</td></tr>
<tr><td><code>>> <<</code></td><td>décalage de bits à droite et à gauche</td></tr>
<tr><td><code>&</code></td><td>ET bit-par-bit</td></tr>
<tr><td><code>| ^</code></td><td>OU, XOR bit-par-bit</td></tr>
<tr><td><code>< <= >= ></code></td><td>compare</td></tr>
<tr><td><code>== <> !=</code></td><td>test d'égalité ou de différence</td></tr>
<tr><td><code>and</code></td><td>ET logique</td></tr>
<tr><td><code>or</code></td><td>OU logique</td></tr>
</table>

Si vous le préférez, les symbols alternatifs suivants peuvent être utilisés:

<table>
<tr><td><code>div mod</code></td><td>peuvent remplacer / et %</td></tr>
<tr><td><code>! && ||</code></td><td>peuvent remplacer <code>not</code>, <code>and</code>, <code>or</code></td></tr>
</table>

Les opérateurs de comparaison et les opérateurs logiques sont destinés à être utilisés avec l'opérateur d'évaluation conditionnelle:

```
compute "($temp &gt; 0 ? log($temp) : -9999)"
```

Les constantes et fonctions mathématiques classiques sont disponibles aussi:

`pi e`	les constantes universelles
`cos sin tan`	fonctions trigonométriques
`acos asin atan atan2`	fonctions trigonométriques inverses
`cosh sinh tanh`	fonctions hyperboliques
`exp log log10 pow sqr sqrt`	puissance et fonctions logarithmiques
`ceil floor round frac fmod`	fonctions d'arrondi
`fabs min max isnan isinf`	fonctions de numération

Les calculs sont effectués avec des nombres à virgule flottante encodés sur 32 bits. Les opérations bit à bit (| & >> etc.) sont effectuées sur des entiers 32 bits.

ASSERT

La commande *assert* permet de vérifier si une condition est remplie avant de continuer l'exécution de la tâche. Elle accepte une expression arithmétique en argument, et stoppe l'exécution de la tâche si le résultat de l'expression est FAUX.

Comme pour la commande *compute*, si l'expression contient une erreur de syntaxe ou fait référence à une variable non définie, la tâche sera aussi arrêtée, avec un message d'erreur dans les logs du module. Par contre, il est possible de vérifier si une variable a été définie sans générer de message d'erreur en utilisant la fonction `isset()`.

```
assert    !isset($init_done)
writeline @1C:0004
compute   $init_done = 1
```

Notez que la commander ASSERT sera très utile pour coder des machines à états.

WAIT

La commande *wait* permet d'attendre un certain nombre de millisecondes avant de passer à la commande suivante.

LOG

La commande *log* permet d'afficher une chaîne de caractères dans les log du Yocto-SDI12.

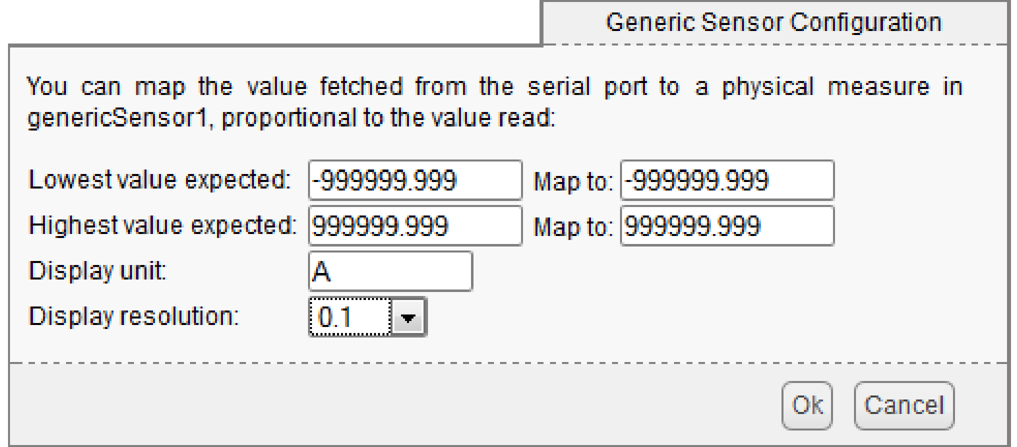
SETPOWER

La commande *setPower* permet de commander automatiquement l'état de la sortie d'alimentation du module via une tâche, par exemple pour mettre en/hors tension un capteur externe.

Si, pour une raison ou une autre, une commande génère une erreur, vous trouverez une trace de cette erreur dans les logs du Yocto-SDI12.

6.4. Les fonctions genericSensor

Le Yocto-SDI12 dispose de 9 fonctions *genericSensor*, dont les valeurs peuvent être librement attribuées par les jobs qui s'exécutent sur le module. Ces fonctions *genericSensor* peuvent être directement accédées depuis l'API Yoctopuce à l'aide de la classe *YGenericSensor*. Elles peuvent aussi être configurées afin d'ajuster leur comportement à la nature des valeurs reportées.



Fenêtre de configuration d'un genericSensor

Unité

Il est possible de définir dans quel système de mesure est spécifié le valeur stockée par le genericSensor.

Précision

Il est possible de définir avec quelle précision doit être représenté la valeur reportée par le genericSensor.

Mapping

Il est possible d'appliquer automatiquement une transformation linéaire aux valeurs stockées dans un *genericSensor*. Imaginons un convertisseur AD qui transmettrait des valeurs entre 0 et 65535 pour des mesures entre 0 et 10V. Il est possible de demander à la fonction *genericSensor* de faire automatiquement la conversion inverse comme illustré ci-dessous.

Exemple de conversion linéaire

Ce mécanisme peut aussi s'avérer très utile pour faire des conversions automatiques, par exemple transformer des pieds en mètres.

Notez que dans le cas du Yocto-SDI12, ce mécanisme générique, que l'on retrouve sur de nombreuses interfaces Yoctopuce, fait double emploi avec la commande *compute* des jobs.

6.5. Exemple de configuration

Voici deux exemples de jobs pour interfacer des capteurs du commerce.

Capteur True TDR-315L (Acclima)

Le capteur True TDR-315L d'Acclima permet de mesurer la teneur en eau, la température, la permittivité du sol ainsi que la conductivité électrique du sol et la conductivité électrique de l'eau dans le sol. Le capteur True TDR-315L permet de mesurer des valeurs dans plusieurs types de sol. Dans notre exemple, nous supposons la mesure est réalisée dans un sol sableux. Une simple tâche périodique permet de l'interfacer: il suffit d'envoyer une commande **M** au capteur, de lire sa réponse et d'affecter les valeurs reçues sur les *genericSensors*. Pour faire cela, on utilise une commande de *writeLine* suivie d'une commande *expect*. On suppose que l'adresse du capteur est '1'.

- Tâche 1 (périodique, 30000ms)
  - `writeLine 1M!`
  - `expect 1:($1:FLOAT),($2:FLOAT),($3:FLOAT),($4:FLOAT),($5:FLOAT).*`

Capteur WET150 (Delta-t devices)

Le capteur WET150 de *Delta-t devices* permet de mesurer la teneur en eau, la température, la permittivité du sol ainsi que la conductivité électrique du sol et la conductivité électrique de l'eau dans le sol. Le capteur WET150 permet de mesurer des valeurs dans plusieurs types de sol, dans notre exemple le mesure est réalisée dans un sol de type "organic". D'après la documentation du capteur, la commande de mesure pour un sol *organic* est **aM2!**. Pour cet exemple, deux capteurs identiques avec les adresses '0' et '2' sont utilisés. Seules les valeurs de température et de teneur en eau seront affectées sur les *genericSensors*. Pour faire cela on utilise deux tâches, une tâche par capteur. Les deux tâches sont assez similaires: envoi de la commande **M2!** suivi d'une commande expect pour affecter les valeurs reçues.

- Tâche 1 (périodique, 30000ms)
  - `writeLine 0M2!`
  - `expect 0:($1:FLOAT),(FLOAT),($2:FLOAT),(FLOAT),(FLOAT).*`
- Tâche 2 (périodique, 30000ms)
  - `writeLine 2M2!`
  - `expect 2:($3:FLOAT),(FLOAT),($4:FLOAT),(FLOAT),(FLOAT).*`

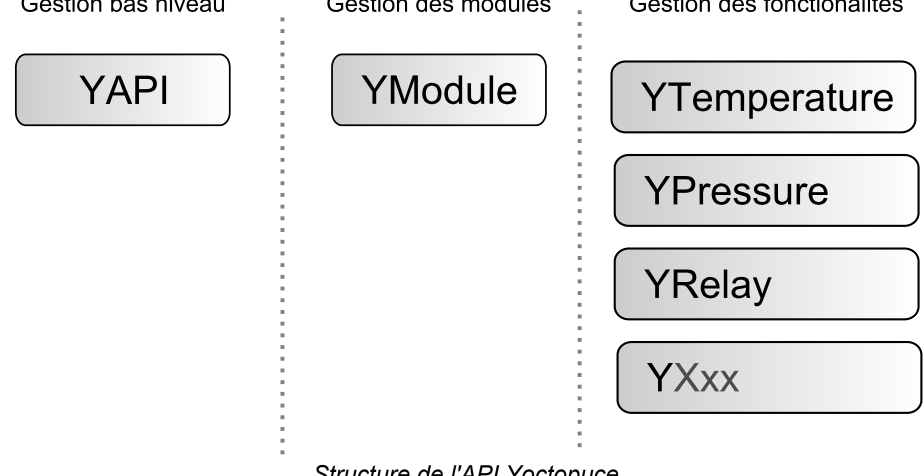
7. Programmation, concepts généraux

L'API Yoctopuce a été pensée pour être à la fois simple à utiliser, et suffisamment générique pour que les concepts utilisés soient valables pour tous les modules de la gamme Yoctopuce et ce dans tous les langages de programmation disponibles. Ainsi, une fois que vous aurez compris comment piloter votre Yocto-SDI12 dans votre langage de programmation favori, il est très probable qu'apprendre à utiliser un autre module, même dans un autre langage, ne vous prendra qu'un minimum de temps.

7.1. Paradigme de programmation

L'API Yoctopuce est une API orientée objet. Mais, dans un souci de simplicité, seules les bases de la programmation objet ont été utilisées. Même si la programmation objet ne vous est pas familière, il est peu probable que cela vous soit un obstacle à l'utilisation des produits Yoctopuce. Notez que vous n'aurez jamais à allouer ou désallouer un objet lié à l'API Yoctopuce: cela est géré automatiquement.

Il existe une classe par type de fonctionnalité Yoctopuce. Le nom de ces classes commence toujours par un Y suivi du nom de la fonctionnalité, par exemple *YTemperature*, *YRelay*, *YPressure*, etc.. Il existe aussi une classe *YModule*, dédiée à la gestion des modules en temps que tels, et enfin il existe la classe statique YAPI, qui supervise le fonctionnement global de l'API et gère les communications à bas niveau.



Structure de l'API Yoctopuce.

La classe YSensor

A chaque fonctionnalité d'un module Yoctopuce, correspond une classe: YTemperature pour mesurer la température, YVoltage pour mesurer une tension, YRelay pour contrôler un relais, etc. Il existe cependant une classe spéciale qui peut faire plus: YSensor.

Cette classe YSensor est la classe parente de tous les senseurs Yoctopuce, elle permet de contrôler n'importe quel senseur, quel que soit son type, en donnant accès au fonctions communes à tous les senseurs. Cette classe permet de simplifier la programmation d'applications qui utilisent beaucoup de senseurs différents. Mieux encore, si vous programmez une application basée sur la classe YSensor, elle sera compatible avec tous les senseurs Yoctopuce, y compris ceux qui n'existent pas encore.

Programmation

Dans l'API Yoctopuce, la priorité a été mise sur la facilité d'accès aux fonctionnalités des modules en offrant la possibilité de faire abstraction des modules qui les implémentent. Ainsi, il est parfaitement possible de travailler avec un ensemble de fonctionnalités sans jamais savoir exactement quel module les héberge au niveau matériel. Cela permet de considérablement simplifier la programmation de projets comprenant un nombre important de modules.

Du point de vue programmation, votre Yocto-SDI12 se présente sous la forme d'un module hébergeant un certain nombre de fonctionnalités. Dans l'API , ces fonctionnalités se présentent sous la forme d'objets qui peuvent être retrouvés de manière indépendante, et ce de plusieurs manières.

Accès aux fonctionnalités d'un module

Accès par nom logique

Chacune des fonctionnalités peut se voir assigner un nom logique arbitraire et persistant: il restera stocké dans la mémoire flash du module, même si ce dernier est débranché. Un objet correspondant à une fonctionnalité *Xxx* munie d'un nom logique pourra ensuite être retrouvée directement à l'aide de ce nom logique et de la méthode *YXxx.FindXxx*. Notez cependant qu'un nom logique doit être unique parmi tous les modules connectés.

Accès par énumération

Vous pouvez énumérer toutes les fonctionnalités d'un même type sur l'ensemble des modules connectés à l'aide des fonctions classiques d'énumération *FirstXxx* et *nextXxxx* disponibles dans chacune des classes *YXxx*.

Accès par nom hardware

Chaque fonctionnalité d'un module dispose d'un nom hardware, assigné en usine qui ne peut être modifié. Les fonctionnalités d'un module peuvent aussi être retrouvées directement à l'aide de ce nom hardware et de la fonction *YXxx.FindXxx* de la classe correspondante.

Différence entre *Find* et *First*

Les méthodes *YXxx.FindXxxx* et *YXxx.FirstXxxx* ne fonctionnent pas exactement de la même manière. Si aucun module n'est disponible *YXxx.FirstXxxx* renvoie une valeur nulle. En revanche, même si aucun module ne correspond, *YXxx.FindXxxx* renverra objet valide, qui ne sera pas "online" mais qui pourra le devenir, si le module correspondant est connecté plus tard.

Manipulation des fonctionnalités

Une fois l'objet correspondant à une fonctionnalité retrouvé, ses méthodes sont disponibles de manière tout à fait classique. Notez que la plupart de ces sous-fonctions nécessitent que le module hébergeant la fonctionnalité soit branché pour pouvoir être manipulées. Ce qui n'est en général jamais garanti, puisqu'un module USB peut être débranché après le démarrage du programme de contrôle. La méthode *isOnline()*, disponible dans chaque classe, vous sera alors d'un grand secours.

Accès aux modules

Bien qu'il soit parfaitement possible de construire un projet en faisant abstraction de la répartition des fonctionnalités sur les différents modules, ces derniers peuvent être facilement retrouvés à l'aide de l'API. En fait, ils se manipulent d'une manière assez semblable aux fonctionnalités. Ils disposent d'un numéro de série affecté en usine qui permet de retrouver l'objet correspondant à l'aide de *YModule.Find()*. Les modules peuvent aussi se voir affecter un nom logique arbitraire qui permettra de les retrouver ensuite plus facilement. Et enfin la classe *YModule* comprend les méthodes d'énumération *YModule.FirstModule()* et *nextModule()* qui permettent de dresser la liste des modules connectés.

Interaction Function / Module

Du point de vue de l'API, les modules et leurs fonctionnalités sont donc fortement décorrélés à dessein. Mais l'API offre néanmoins la possibilité de passer de l'un à l'autre. Ainsi la méthode *get_module()*, disponible dans chaque classe de fonctionnalité, permet de retrouver l'objet correspondant au module hébergeant cette fonctionnalité. Inversement, la classe *YModule* dispose d'un certain nombre de méthodes permettant d'énumérer les fonctionnalités disponibles sur un module.

7.2. Le module Yocto-SDI12

Le module Yocto-SDI12 est une interface série TTL autonome, avec datalogger intégré.

module : Module

attribut	type	modifiable ?
productName	Texte	lecture seule
serialNumber	Texte	lecture seule
logicalName	Texte	modifiable
productId	Entier (hexadécimal)	lecture seule
productRelease	Entier (hexadécimal)	lecture seule
firmwareRelease	Texte	lecture seule
persistentSettings	Type énuméré	modifiable
luminosity	0..100%	modifiable
beacon	On/Off	modifiable
upTime	Temps	lecture seule
usbCurrent	Courant consommé (en mA)	lecture seule
rebootCountdown	Nombre entier	modifiable
userVar	Nombre entier	modifiable

sdi12Port : Sdi12Port

attribut	type	modifiable ?
logicalName	Texte	modifiable
advertisedValue	Texte	modifiable
rxCount	Nombre entier	lecture seule
txCount	Nombre entier	lecture seule
errCount	Nombre entier	lecture seule
rxMsgCount	Nombre entier	lecture seule
txMsgCount	Nombre entier	lecture seule
lastMsg	Texte	lecture seule
currentJob	Texte	modifiable
startupJob	Texte	modifiable
jobMaxTask	Nombre entier	lecture seule
jobMaxSize	Nombre entier	lecture seule
command	Texte	modifiable
protocol	Type de protocole de communication	modifiable
voltageLevel	Type énuméré	modifiable
serialMode	Paramètres de transmission série	modifiable

genericSensor1 : GenericSensor
genericSensor2 : GenericSensor
genericSensor3 : GenericSensor
genericSensor4 : GenericSensor
genericSensor5 : GenericSensor
genericSensor6 : GenericSensor
genericSensor7 : GenericSensor
genericSensor8 : GenericSensor
genericSensor9 : GenericSensor
genericSensor10 : GenericSensor

attribut	type	modifiable ?
`logicalName`	Texte	modifiable
`advertisedValue`	Texte	modifiable
`unit`	Texte	modifiable
`currentValue`	Nombre (virgule fixe)	lecture seule
`lowestValue`	Nombre (virgule fixe)	modifiable
`highestValue`	Nombre (virgule fixe)	modifiable
`currentRawValue`	Nombre (virgule fixe)	lecture seule
`logFrequency`	Fréquence	modifiable
`reportFrequency`	Fréquence	modifiable
`advMode`	Type énuméré	modifiable
`calibrationParam`	Paramètrs de calibration	modifiable
`resolution`	Nombre (virgule fixe)	modifiable
`sensorState`	Nombre entier	lecture seule
`signalValue`	Nombre (virgule fixe)	lecture seule
`signalUnit`	Texte	lecture seule
`signalRange`	Plage de valeurs	modifiable
`valueRange`	Plage de valeurs	modifiable
`signalBias`	Nombre (virgule fixe)	modifiable
`signalSampling`	Type énuméré	modifiable
`enabled`	Booléen	modifiable

dataLogger : DataLogger

attribut	type	modifiable ?
`logicalName`	Texte	modifiable
`advertisedValue`	Texte	modifiable
`currentRunIndex`	Nombre entier	lecture seule
`timeUTC`	Heure UTC	modifiable
`recording`	Type énuméré	modifiable
`autoStart`	On/Off	modifiable
`beaconDriven`	On/Off	modifiable
`usage`	0..100%	lecture seule
`clearHistory`	Booléen	modifiable

files : Files

attribut	type	modifiable ?
`logicalName`	Texte	modifiable
`advertisedValue`	Texte	modifiable
`filesCount`	Nombre entier	lecture seule
`freeSpace`	Nombre entier	lecture seule

7.3. Module

Interface de contrôle des paramètres généraux des modules Yoctopuce

La classe `YModule` est utilisable avec tous les modules USB de Yoctopuce. Elle permet de contrôler les paramètres généraux du module, et d'énumérer les fonctions fournies par chaque module.

productName

Chaîne de caractères contenant le nom commercial du module, préprogrammé en usine.

serialNumber

Chaîne de caractères contenant le numéro de série, unique et préprogrammé en usine. Pour un module Yocto-SDI12, ce numéro de série commence toujours par YSDIMK01. Il peut servir comme point de départ pour accéder par programmation à un module particulier.

logicalName

Chaîne de caractères contenant le nom logique du module, initialement vide. Cet attribut peut être changé au bon vouloir de l'utilisateur. Une fois initialisé à une valeur non vide, il peut servir de point de départ pour accéder à un module particulier. Si deux modules avec le même nom logique se trouvent sur le même montage, il n'y a pas moyen de déterminer lequel va répondre si l'on tente un accès par ce nom logique. Le nom logique du module est limité à 19 caractères parmi `A..Z`,`a..z`,`0..9`,`_` et `-`.

productId

Identifiant USB du module, préprogrammé à la valeur 171 en usine.

productRelease

Numéro de révision du module hardware, préprogrammé en usine. La révision originale du retourne la valeur 1, la révision B retourne la valeur 2, etc.

firmwareRelease

Version du logiciel embarqué du module, elle change à chaque fois que le logiciel embarqué est mis à jour.

persistentSettings

Etat des réglages persistants du module: chargés depuis la mémoire non-volatile, modifiés par l'utilisateur ou sauvegardés dans la mémoire non volatile.

luminosity

Intensité lumineuse maximale des leds informatives (comme la Yocto-Led) présentes sur le module. C'est une valeur entière variant entre 0 (leds éteintes) et 100 (leds à l'intensité maximum). La valeur par défaut est 50. Pour changer l'intensité maximale des leds de signalisation du module, ou les éteindre complètement, il suffit donc de modifier cette valeur.

beacon

Etat de la balise de localisation du module.

upTime

Temps écoulé depuis la dernière mise sous tension du module.

usbCurrent

Courant consommé par le module sur le bus USB, en milli-ampères.

rebootCountdown

Compte à rebours pour déclencher un redémarrage spontané du module.

userVar

Attribut de type entier 32 bits à disposition de l'utilisateur.

7.4. Sdi12Port

Interface pour intéragir avec les ports SDI12

La classe `YSdi12Port` permet de piloter entièrement un module d'interface SDI12 Yoctopuce. Elle permet d'envoyer et de recevoir des données, et de configurer les paramètres de transmission (vitesse, nombre de bits, parité, contrôle de flux et protocole). Notez que les interfaces SDI12 Yoctopuce ne sont pas visibles comme des ports COM virtuels. Ils sont faits pour être utilisés comme tous les autres modules Yoctopuce.

logicalName

Chaîne de caractères contenant le nom logique du port SDI12, initialement vide. Cet attribut peut être changé au bon vouloir de l'utilisateur. Un fois initialisé à une valeur non vide, il peut servir de point de départ pour accéder à directement au port SDI12. Si deux ports SDI12 portent le même nom logique dans un projet, il n'y a pas moyen de déterminer lequel va répondre si l'on tente un accès par ce nom logique. Le nom logique du module est limité à 19 caractères parmi `A..Z`,`a..z`,`0..9`,`_` et `-`.

advertisedValue

Courte chaîne de caractères résumant l'état actuel du port SDI12, et qui sera publiée automatiquement jusqu'au hub parent. Pour un port SDI12, la valeur publiée est une chaîne hexadécimale qui change à chaque caractère reçu. Elle est composée des 16 bits inférieur du compte de caractères reçus, et du code ASCII du dernier caractère reçu.

rxCount

Nombre d'octets reçus depuis la dernière mise à zéro.

txCount

Nombre d'octets transmis depuis la dernière mise à zéro.

errCount

Nombre d'erreurs de communication détectées depuis la dernière mise à zéro.

rxMsgCount

Nombre de messages reçus depuis la dernière mise à zéro.

txMsgCount

Nombre de messages transmis depuis la dernière mise à zéro.

lastMsg

Dernier message reçu (pour les protocoles de type Line, Frame et Modbus).

currentJob

Nom du fichier de tâches actif.

startupJob

Nom du fichier de tâches à exécuter au démarrage du module.

jobMaxTask

Nombre maximal de tâches dans un job supporté par le module.

jobMaxSize

Taille maximale d'un fichier job.

command

Attribut magique permettant d'envoyer des commandes au port série. Si une commande n'est pas interprétée comme attendue, consultez les logs du module.

protocol

Type de protocole utilisé sur la communication série.

voltageLevel

Niveau de voltage utilisé sur la ligne série.

serialMode

Taux de transfert, nombre de bits, parité et bits d'arrêt.

7.5. GenericSensor

Interface pour intéragir avec les capteurs de type GenericSensor, disponibles par exemple dans le Yocto-0-10V-Rx, le Yocto-4-20mA-Rx, le Yocto-Bridge et le Yocto-milliVolt-Rx

La classe `YGenericSensor` permet de lire et de configurer les transducteurs de signaux Yoctopuce. Elle hérite de la classe `YSensor` toutes les fonctions de base des capteurs Yoctopuce: lecture de mesures, callbacks, enregistreur de données. De plus, elle permet de configurer une conversion automatique entre le signal mesuré et la grandeur physique représentée.

logicalName

Chaîne de caractères contenant le nom logique du capteur générique, initialement vide. Cet attribut peut être changé au bon vouloir de l'utilisateur. Un fois initialisé à une valeur non vide, il peut servir de point de départ pour accéder à directement au capteur générique. Si deux capteurs génériques portent le même nom logique dans un projet, il n'y a pas moyen de déterminer lequel va répondre si l'on tente un accès par ce nom logique. Le nom logique du module est limité à 19 caractères parmi `A..Z`,`a..z`,`0..9`,`_` et `-`.

advertisedValue

Courte chaîne de caractères résumant l'état actuel du capteur générique, et qui sera publiée automatiquement jusqu'au hub parent. Pour un capteur générique, la valeur publiée est la valeur courante de la mesure.

unit

Courte chaîne de catactères représentant l'unité dans laquelle la valeur mesurée est exprimée.

currentValue

Valeur actuelle de la mesure, en l'unité spécifiée, sous forme de nombre à virgule.

lowestValue

Valeur minimale de la mesure, en l'unité spécifiée, sous forme de nombre à virgule.

highestValue

Valeur maximale de la mesure, en l'unité spécifiée, sous forme de nombre à virgule.

currentRawValue

Valeur brute mesurée par le capteur (sans arrondi ni calibration), sous forme de nombre à virgule.

logFrequency

Fréquence d'enregistrement des mesures dans le datalogger, ou "OFF" si les mesures ne doivent pas être stockées dans la mémoire de l'enregistreur de données.

reportFrequency

Fréquence de notification périodique des valeurs mesurées, ou "OFF" si les notifications périodiques de valeurs sont désactivées.

advMode

Mode de calcul de la valeur publiée jusqu'au hub parent (advertisedValue).

calibrationParam

Paramètres de calibration supplémentaires (par exemple pour compenser l'effet d'un boîtier), sous forme de tableau d'entiers 16 bit.

resolution

Résolution de la mesure (précision de la représentation, mais pas forcément de la mesure elle-même).

sensorState

Etat du capteur (zero lorsque qu'une mesure actuelle est disponible).

signalValue

Valeur actuelle du signal électrique mesuré par le capteur, sous forme de nombre à virgule.

signalUnit

Courte chaîne de catactères représentant l'unité du signal électrique utilisé par le capteur.

signalRange

Plage de valeurs électriques utilisées par le capteur.

valueRange

Plage de valeurs physiques mesurées par le capteur, utilisée pour la conversion du signal.

signalBias

Biais du signal électrique pour la correction du point zéro.

signalSampling

Méthode d'échantillonnage du signal à utiliser.

enabled

Activation/désactivation de la mesure.

7.6. DataLogger

Interface de contrôle de l'enregistreur de données, présent sur la plupart des capteurs Yoctopuce.

La plupart des capteurs Yoctopuce sont équipés d'une mémoire non-volatile. Elle permet de mémoriser les données mesurées d'une manière autonome, sans nécessiter le suivi permanent d'un ordinateur. La classe `YDataLogger` contrôle les paramètres globaux de cet enregistreur de données. Le contrôle de l'enregistrement (start / stop) et la récupération des données se fait au niveau des objets qui gèrent les senseurs.

logicalName

Chaîne de caractères contenant le nom logique de l'enregistreur de données, initialement vide. Cet attribut peut être changé au bon vouloir de l'utilisateur. Un fois initialisé à une valeur non vide, il peut servir de point de départ pour accéder à directement à l'enregistreur de données. Si deux enregistreurs de données portent le même nom logique dans un projet, il n'y a pas moyen de déterminer lequel va répondre si l'on tente un accès par ce nom logique. Le nom logique du module est limité à 19 caractères parmi `A..Z,a..z,0..9,_` et `-`.

advertisedValue

Courte chaîne de caractères résumant l'état actuel de l'enregistreur de données, et qui sera publiée automatiquement jusqu'au hub parent. Pour un enregistreur de données, la valeur publiée est son état d'activation (ON ou OFF).

currentRunIndex

Numéro du Run actuel, correspondant au nombre de fois que le module a été mis sous tension avec la fonction d'enregistreur de données active.

timeUTC

Heure UTC courante, lorsque l'on désire associer une référence temporelle absolue aux données enregistrées. Cette heure doit être configurée explicitement par logiciel.

recording

Etat d'activité de l'enregistreur de données. L'enregistreur peut être activé ou désactivé à volonté par cet attribut, mais son état à la mise sous tension est déterminé par l'attribut persistent **autoStart**. Lorsque l'enregistreur est enclenché mais qu'il n'est pas encore prêt pour enregistrer, son état est PENDING.

autoStart

Activation automatique de l'enregistreur de données à la mise sous tension. Cet attribut permet d'activer systématiquement l'enregistreur à la mise sous tension, sans devoir l'activer par une commande logicielle. Attention si le module n'a pas de source de temps à sa disposition, il va attendre environ 8 sec avant de démarrer automatiquement l'enregistrement

beaconDriven

Permet de synchroniser l?état de la balise de localisation avec l?état de l'enregistreur de données. Quand cet attribut est activé il est possible de démarrer et arrêter l'enregistrement en utilisant le Yocto-bouton du module ou l?attribut `beacon` de la fonction YModule. De la même manière si l'attribut `recording` de la fonction datalogger est modifié, l?état de la balise de localisation est mis à jour. Note: quand cet attribut est activé balise de localisation du module clignote deux fois plus lentement.

usage

Pourcentage d'utilisation de la mémoire d'enregistrement.

clearHistory

Attribut qui peut être mis à vrai pour effacer l'historique des mesures.

7.7. Files

Interface pour intéragir avec les systèmes de fichier, disponibles par exemple dans le Yocto-Color-V2, le Yocto-SPI, le YoctoHub-Ethernet et le YoctoHub-GSM-4G

La class YFiles permet d'accéder au système de fichier embarqué sur certains modules Yoctopuce. Le stockage de fichiers permet par exemple de personnaliser un service web (dans le cas d'un

module connecté au réseau) ou pour d'ajouter un police de caractères (dans le cas d'un module d'affichage).

logicalName

Chaîne de caractères contenant le nom logique du système de fichier, initialement vide. Cet attribut peut être changé au bon vouloir de l'utilisateur. Un fois initialisé à une valeur non vide, il peut servir de point de départ pour accéder à directement au système de fichier. Si deux systèmes de fichier portent le même nom logique dans un projet, il n'y a pas moyen de déterminer lequel va répondre si l'on tente un accès par ce nom logique. Le nom logique du module est limité à 19 caractères parmi `A..Z`,`a..z`,`0..9`,`_` et `-`.

advertisedValue

Courte chaîne de caractères résumant l'état actuel du système de fichier, et qui sera publiée automatiquement jusqu'au hub parent. Pour un système de fichier, la valeur publiée est le nombre de fichiers présents.

filesCount

Nombre de fichiers présents dans le système de fichier.

freeSpace

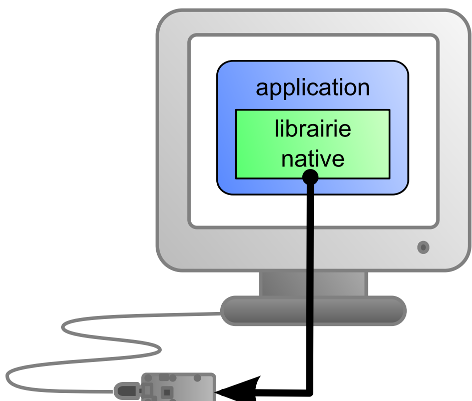
Espace disponible dans le système de fichiers pour charger des nouveaux fichiers, en octets.

7.8. Quelle interface: Native, DLL ou Service?

Il y existe plusieurs méthodes pour contrôler un module USB Yoctopuce depuis un programme.

Contrôle natif

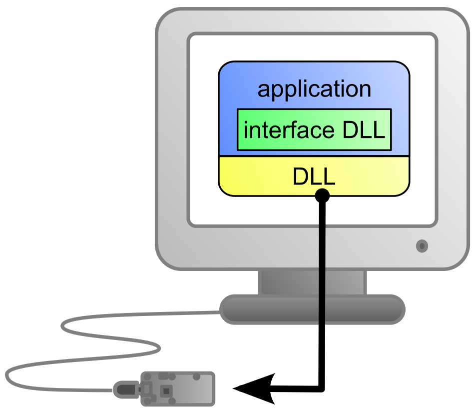
Dans ce cas de figure le programme pilotant votre projet est directement compilé avec une librairie qui offre le contrôle des modules. C'est objectivement la solution la plus simple et la plus élégante pour l'utilisateur final. Il lui suffira de brancher le câble USB et de lancer votre programme pour que tout fonctionne. Malheureusement, cette technique n'est pas toujours disponible ou même possible.



L'application utilise la librairie native pour contrôler le module connecté en local

Contrôle natif par DLL

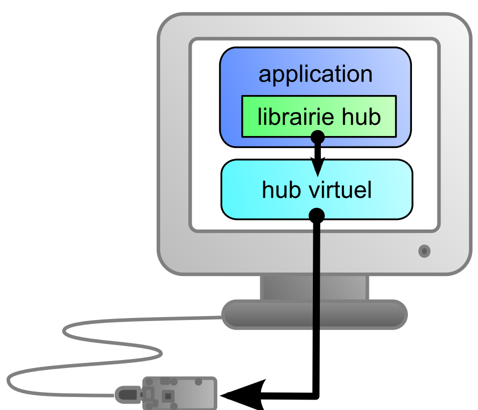
Ici l'essentiel du code permettant de contrôler les modules se trouve dans une DLL, et le programme est compilé avec une petite librairie permettant de contrôler cette DLL. C'est la manière la plus rapide pour coder le support des modules dans un language particulier. En effet la partie "utile" du code de contrôle se trouve dans la DLL qui est la même pour tous les langages, offrir le support pour un nouveau langage se limite à coder la petite librairie qui contrôle la DLL. Du point de de l'utilisateur final, il y a peu de différences: il faut simplement être sûr que la DLL sera installée sur son ordinateur en même temps que le programme principal.



L'application utilise la DLL pour contrôler nativement le module connecté en local

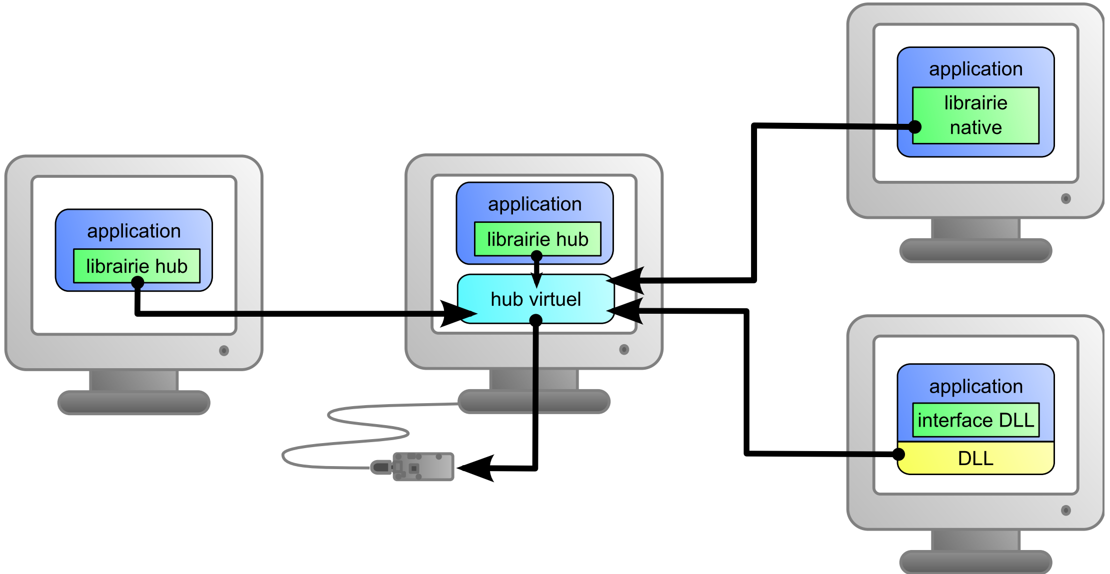
Contrôle par un service

Certains langages ne permettent tout simplement pas d'accéder facilement au niveau matériel de la machine. C'est le cas de Javascript par exemple. Pour gérer ce cas, Yoctopuce offre la solution sous la forme d'un petit service, appelé VirtualHub qui, lui, est capable d'accéder aux modules, et votre application n'a plus qu'à utiliser une librairie qui offrira toutes les fonctions nécessaires au contrôle des modules en passant par l'intermédiaire de ce VirtualHub. L'utilisateur final se verra obligé de lancer VirtualHub avant de lancer le programme de contrôle du projet proprement dit, à moins qu'il ne décide d'installer VirtualHub sous la forme d'un service/démon, auquel cas VirtualHub se lancera automatiquement au démarrage de la machine..



L'application se connecte au service VirtualHub pour connecter le module.

En revanche la méthode de contrôle par un service offre un avantage non négligeable: l'application n'est pas n'obligée de tourner sur la machine où se trouvent les modules: elle peut parfaitement se trouver sur un autre machine qui se connectera au service pour piloter les modules. De plus, les librairie natives et DLL évoquées plus haut sont aussi capables de se connecter à distance à une ou plusieurs instances de VirtualHub.



Lorsqu'on utilise VirtualHub, l'application de contrôle n'a plus besoin d'être sur la même machine que le module.

Quel que soit langage de programmation choisi et le paradigme de contrôle utilisé, la programmation reste strictement identique. D'un langage à l'autre les fonctions ont exactement le même nom, prennent les mêmes paramètres. Les seules différences sont liées aux contraintes des langages eux-mêmes.

Language	Natif	Natif avec .DLL/.so	VirtualHub
Ligne de commande	✔	-	✔
Python	-	✔	✔
C++	✔	✔	✔
C# .Net	-	✔	✔
C# UWP	✔	-	✔
LabVIEW	-	✔	✔
Java	-	✔	✔
Java pour Android	✔	-	✔
TypeScript	-	-	✔
JavaScript / ECMAScript	-	-	✔
PHP	-	-	✔
VisualBasic .Net	-	✔	✔
Delphi	-	✔	✔
Objective-C	✔	-	✔

Méthode de support pour les différents langages.

7.9. Accéder aux modules à travers un hub

VirtualHub pour contourner la limitation d'accès à USB

Une seule application à la fois peut avoir accès nativement aux modules Yoctopuce. Cette limitation est liée au fait que deux processus différents ne peuvent pas parler en même temps à un périphérique USB. En général, ce type de problème est réglé par un driver qui se charge de faire la police pour éviter que plusieurs processus ne se battent pour le même périphérique. Mais les produits Yoctopuce n'utilisent pas de drivers. Par conséquent, le premier processus qui arrive à accéder au mode natif le garde pour lui jusqu'à ce que `UnregisterHub` ou `FreeApi` soit appelé.

Si votre application essaie de communiquer en mode natif avec les modules Yoctopuce, mais qu'une autre application vous empêche d'y accéder, vous revecrez le message d'erreur suivant:

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

La solution est d'utiliser VirtualHub localement sur votre machine et de vous en servir comme passerelle pour vos applications. Ainsi, si toutes vos application utilisent VirtualHub, vous n'aurez plus de conflit et vous pourrez accéder en tout temps à tous vos modules.

Pour passer du mode natif au mode réseau sur votre machine locale, il vous suffit de changer le paramètre de l'appel à `YAPI.RegisterHub` et d'indiquer `127.0.0.1` à la place de `usb`.

Avec un YoctoHub

Un YoctoHub se comporte exactement comme un ordinateur faisant tourner VirtualHub. La seule différence entre un programme utilisant l'API Yoctopuce utilisant des modules en USB natif et ce même programme utilisant des modules Yoctopuce connectés à un YoctoHub se situe au niveau de l'appel à `RegisterHub`. Pour utiliser des modules USB connectés en natif, le paramètre de `RegisterHub` est `usb`, pour utiliser des modules connectés à un YoctoHub, il suffit de remplacer ce paramètre par l'adresse IP du YoctoHub.

Il y a donc trois cas de figure: le mode natif, le mode réseau à travers VirtualHub sur votre machine locale, ou à travers un YoctoHub. Pour passer de l'un à l'autre, il vous suffit de changer le paramètre de l'appel à `YAPI.RegisterHub` comme dans les exemples ci-dessous:

```
YAPI.RegisterHub("usb",errmsg);  // utilisation en mode natif USB

YAPI.RegisterHub("127.0.0.1",errmsg);  // utilisation en mode réseau local avec VirtualHub

YAPI.RegisterHub("192.168.0.10",errmsg); // utilisation avec YoctoHub dont l'adresse IP est
192.168.0.10
```

7.10. Programmation, par où commencer?

Arrivé à ce point du manuel, vous devriez connaître l'essentiel de la théorie à propos de votre Yocto-SDI12. Il est temps de passer à la pratique. Il vous faut télécharger la librairie Yoctopuce pour votre language de programmation favori depuis le site web de Yoctopuce[1]. Puis sautez directement au chapitre correspondant au langage de programmation que vous avez choisi.

Tous les exemples décrits dans ce manuel sont présents dans les librairies de programmation. Dans certains langages, les librairies comprennent aussi quelques applications graphiques complètes avec leur code source.

Une fois que vous maîtriserez la programmation de base de votre module, vous pourrez vous intéresser au chapitre concernant la programmation avancée qui décrit certaines techniques qui vous permettront d'exploiter au mieux votre Yocto-SDI12.

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

8. Utilisation du Yocto-SDI12 en ligne de commande

Lorsque vous désirez effectuer une opération ponctuelle sur votre Yocto-SDI12, comme la lecture d'une valeur, le changement d'un nom logique, etc.. vous pouvez bien sur utiliser VirtualHub, mais il existe une méthode encore plus simple, rapide et efficace: l'API en ligne de commande.

L'API en ligne de commande se présente sous la forme d'un ensemble d'exécutables, un par type de fonctionnalité offerte par l'ensemble des produits Yoctopuce. Ces exécutables sont fournis pré-compilés pour toutes les plateformes/OS officiellement supportés par Yoctopuce. Bien entendu, les sources de ces exécutables sont aussi fournies[1].

8.1. Installation

Téléchargez l'API en ligne de commande[2]. Il n'y a pas de programme d'installation à lancer, copiez simplement les exécutables correspondant à votre plateforme/OS dans le répertoire de votre choix. Ajoutez éventuellement ce répertoire à votre variable environnement PATH pour avoir accès aux exécutables depuis n'importe où. C'est tout, il ne vous reste plus qu'à brancher votre Yocto-SDI12, ouvrir un shell et commencer à travailler en tapant par exemple:

```
C:\>YSdi12Port any set_voltageLevel VOLTAGELEVEL_SDI12
C:\>YSdi12Port any discoverSingleSensor
C:\>YSdi12Port any readSensor 1 M 500
```

Sous Linux, pour utiliser l'API en ligne de commande, vous devez soit être root, soit définir une règle *udev* pour votre système. Vous trouverez plus de détails au chapitre *Problèmes courants*.

8.2. Utilisation: description générale

Tous les exécutables de l'API en ligne de commande fonctionnent sur le même principe: ils doivent être appelés de la manière suivante:

```
C:\>Executable [options] [cible] commande [paramètres]
```

Les `[options]` gèrent le fonctionnement global des commandes , elles permettent par exemple de piloter des modules à distance à travers le réseau, ou encore elles peuvent forcer les modules à sauver leur configuration après l'exécution de la commande.

[1] Si vous souhaitez recompiler l'API en ligne de commande, vous aurez aussi besoin de l'API C++
[2] http://www.yoctopuce.com/FR/libraries.php

La `[cible]` est le nom du module ou de la fonction auquel la commande va s'appliquer. Certaines commandes très génériques n'ont pas besoin de cible. Vous pouvez aussi utiliser les alias "*any*" ou "*all*", ou encore une liste de noms, séparés par des virgules, sans espace.

La `commande` est la commande que l'on souhaite exécuter. La quasi-totalité des fonctions disponibles dans les API de programmation classiques sont disponibles sous forme de commandes. Vous n'êtes pas obligé des respecter les minuscules/majuscules et les caractères soulignés dans le nom de la commande.

Les `[paramètres]` sont, assez logiquement, les paramètres dont la commande a besoin.

A tout moment les exécutables de l'API en ligne de commande sont capables de fournir une aide assez détaillée: Utilisez par exemple

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

pour connaître la liste de commandes disponibles pour un exécutable particulier de l'API en ligne de commande, ou encore:

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

Pour obtenir une description détaillée des paramètres d'une commande.

8.3. Contrôle de la fonction Sdi12Port

Pour contrôler la fonction Sdi12Port de votre Yocto-SDI12, vous avez besoin de l'exécutable YSdi12Port.

Vous pouvez par exemple lancer:

```
C:\>YSdi12Port any set_voltageLevel VOLTAGELEVEL_SDI12
C:\>YSdi12Port any discoverSingleSensor
C:\>YSdi12Port any readSensor 1 M 500
```

Cet exemple utilise la cible "*any*" pour signifier que l'on désire travailler sur la première fonction Sdi12Port trouvée parmi toutes celles disponibles sur les modules Yoctopuce accessibles au moment de l'exécution. Cela vous évite d'avoir à connaître le nom exact de votre fonction et celui de votre module.

Mais vous pouvez tout aussi bien utiliser des noms logiques que vous auriez préalablement configurés. Imaginons un module Yocto-SDI12 avec le numéros de série *YSDIMK01-123456* que vous auriez appelé "*MonModule*" et dont vous auriez nommé la fonction *sdi12Port* "*MaFonction*", les cinq appels suivants seront strictement équivalents (pour autant que *MaFonction* ne soit définie qu'une fois, pour éviter toute ambiguïté).

```
C:\>YSdi12Port YSDIMK01-123456.sdi12Port describe

C:\>YSdi12Port YSDIMK01-123456.MaFonction describe

C:\>YSdi12Port MonModule.sdi12Port describe

C:\>YSdi12Port MonModule.MaFonction describe

C:\>YSdi12Port MaFonction describe
```

Pour travailler sur toutes les fonctions Sdi12Port à la fois, utilisez la cible "*all*".

```
C:\>YSdi12Port all describe
```

Pour plus de détails sur les possibilités de l'exécutable`YSdi12Port`, utilisez:

```
C:\>YSdi12Port /help
```

8.4. Contrôle de la partie module

Chaque module peut être contrôlé d'une manière similaire à l'aide de l'exécutable `YModule`. Par exemple, pour obtenir la liste de tous les modules connectés, utilisez:

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

Vous pouvez aussi utiliser la commande suivante pour obtenir une liste encore plus détaillée des modules connectés:

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

Chaque propriété `xxx` du module peut être obtenue grâce à une commande du type `get_xxxx()`, et les propriétés qui ne sont pas en lecture seule peuvent être modifiées à l'aide de la commande `set_xxx()`. Par exemple:

```
C:\>YModule YSDIMK01-12346 set_logicalName MonPremierModule

C:\>YModule YSDIMK01-12346 get_logicalName
```

Modifications des réglages du module

Lorsque que vous souhaitez modifier les réglages d'un module, il suffit d'utiliser la commande `set_xxx` correspondante, cependant cette modification n'a lieu que dans la mémoire vive du module: si le module redémarre, les modifications seront perdues. Pour qu'elle soient mémorisées de manière persistante, il est nécessaire de demander au module de sauvegarder sa configuration courante dans sa mémoire non volatile. Pour cela il faut utiliser la commande `saveToFlash`. Inversement il est possible de forcer le module à oublier ses réglages courants en utilisant la méthode `revertFromFlash`. Par exemple:

```
C:\>YModule YSDIMK01-12346 set_logicalName MonPremierModule
C:\>YModule YSDIMK01-12346 saveToFlash
```

Notez que vous pouvez faire la même chose en seule fois à l'aide de l'option `-s`

```
C:\>YModule -s  YSDIMK01-12346 set_logicalName MonPremierModule
```

Attention, le nombre de cycles d'écriture de la mémoire non volatile du module est limité. Passé cette limite plus rien ne garantit que la sauvegarde des réglages se passera correctement. Cette limite, liée à la technologie employée par le micro-processeur du module se situe aux alentour de 100000 cycles. Pour résumer vous ne pouvez employer la commande `saveToFlash` que 100000 fois au cours de la vie du module. Veillez donc à ne pas appeler cette commande depuis l'intérieur d'une boucle.

8.5. Limitations

L'API en ligne de commande est sujette à la même limitation que les autres API: il ne peut y avoir q'une seule application à la fois qui accède aux modules de manière native. Par défaut l'API en ligne de commande fonctionne en natif.

Cette limitation peut aisément être contournée en utilisant VirtualHub: il suffit de faire tourner VirtualHub[3] sur la machine concernée et d'utiliser les executables de l'API en ligne de commande avec l'option `-r` par exemple, si vous utilisez:

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

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

Vous obtenez un inventaire des modules connectés par USB, en utilisant un accès natif. Si il y a déjà une autre commande en cours qui accède aux modules en natif, cela ne fonctionnera pas. Mais si vous lancez VirtualHub et que vous lancez votre commande sous la forme:

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

cela marchera parce que la commande ne sera plus exécutée nativement, mais à travers VirtualHub. Notez que VirtualHub compte comme une application native.

9. Utilisation du Yocto-SDI12 en Python

Python est un langage interprété orienté objet développé par Guido van Rossum. Il offre l'avantage d'être gratuit et d'être disponible pour la plupart de plate-formes tant Windows qu'Unix. C'est un language idéal pour écrire des petits scripts sur un coin de table. La librairie Yoctopuce est compatible avec Python 2.7 et 3.x jusqu'aux toutes dernières versions officielles. Elle fonctionne sous Windows, macOS et Linux tant Intel qu'ARM. Les interpréteurs Python sont disponibles sur le site de Python [1].

9.1. Fichiers sources

Les classes de la librairie Yoctopuce[2] pour Python que vous utiliserez vous sont fournies au format source. Copiez tout le contenu du répertoire *Sources* dans le répertoire de votre choix et ajoutez ce répertoire à la variable d'environnement *PYTHONPATH*. Si vous utilisez un IDE pour programmer en Python, référez-vous à sa documentation afin le configurer de manière à ce qu'il retrouve automatiquement les fichiers sources de l'API.

9.2. Librairie dynamique

Une partie de la librairie de bas-niveau est écrite en C, mais vous n'aurez a priori pas besoin d'interagir directement avec elle: cette partie est fournie sous forme de DLL sous Windows, de fichier *.so* sous Unix et de fichier *.dylib* sous macOS. Tout a été fait pour que l'interaction avec cette librairie se fasse aussi simplement que possible depuis Python: les différentes versions de la librairie dynamique correspondant aux différents systèmes d'exploitation et architectures sont stockées dans le répertoire *cdll*. L'API va charger automatiquement le bon fichier lors de son initialisation. Vous n'aurez donc pas à vous en soucier.

Si un jour vous deviez vouloir recompiler la librairie dynamique, vous trouverez tout son code source dans la librairie Yoctopuce pour le C++.

Afin de les garder simples, tous les exemples fournis dans cette documentation sont des applications consoles. Il va de soit que que le fonctionnement des librairies est strictement identiques si vous les intégrez dans une application dotée d'une interface graphique.

9.3. Contrôle de la fonction Sdi12Port

[1] http://www.python.org/download/
[2] www.yoctopuce.com/FR/libraries.php

Il suffit de quelques lignes de code pour piloter un Yocto-SDI12. Voici le squelette d'un fragment de code Python qui utilise la fonction Sdi12Port.

```
[...]
# On active la détection des modules sur USB
errmsg=YRefParam()
YAPI.RegisterHub("usb",errmsg)
[...]

# On récupère l'objet permettant d'intéragir avec le module
sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.sdi12Port")

# Pour gérer le hot-plug, on vérifie que le module est là
if sdi12port.isOnline():
    # use sdi12port.set_sdi12Mode()
    [...]

[...]
```

Voyons maintenant en détail ce que font ces quelques lignes.

YAPI.RegisterHub

La fonction `YAPI.RegisterHub` initialise l'API de Yoctopuce en indiquant où les modules doivent être recherchés. Utilisée avec le paramètre `"usb"`, elle permet de travailler avec les modules connectés localement à la machine. Si l'initialisation se passe mal, cette fonction renverra une valeur différente de `YAPI.SUCCESS`, et retournera via l'objet `errmsg` une explication du problème.

YSdi12Port.FindSdi12Port

La fonction `YSdi12Port.FindSdi12Port` permet de retrouver un port SDI12 en fonction du numéro de série de son module hôte et de son nom de fonction. Mais vous pouvez tout aussi bien utiliser des noms logiques que vous auriez préalablement configurés. Imaginons un module Yocto-SDI12 avec le numéros de série *YSDIMK01-123456* que vous auriez appelé "*MonModule*" et dont vous auriez nommé la fonction *sdi12Port* "*MaFonction*", les cinq appels suivants seront strictement équivalents (pour autant que *MaFonction* ne soit définie qu'une fois, pour éviter toute ambiguïté):

```
sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.sdi12Port")
sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.MaFonction")
sdi12port = YSdi12Port.FindSdi12Port("MonModule.sdi12Port")
sdi12port = YSdi12Port.FindSdi12Port("MonModule.MaFonction")
sdi12port = YSdi12Port.FindSdi12Port("MaFonction")
```

`YSdi12Port.FindSdi12Port` renvoie un objet que vous pouvez ensuite utiliser à loisir pour contrôler le port SDI12.

isOnline

La méthode `isOnline()` de l'objet renvoyé par `YSdi12Port.FindSdi12Port` permet de savoir si le module correspondant est présent et en état de marche.

A propos des imports Python

Cette documentation suppose que vous utilisez la librairie Python téléchargée directement depuis le site web de Yoctopuce, mais si vous avez installé la librairie Yoctopuce avec PIP, alors vous devrez préfixer tous les imports avec *yoctopuce.*. Ainsi tous les exemples donnés dans la documentation, tels que:

```
from yocto_api import *
```

doivent être convertis, lorsque que la librairie Yoctopuce a été installée par PIP, en:

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

reset

La méthode `reset()` de l'objet returné par `YSdi12Port.FindSerialPort` vide tous les tampons du port série.

discoverSingleSensor

La méthode `discoverSingleSensor()` cherche l'adresse du capteur connecté sur le port SDI-12 et renvoie un objet avec toute les informations du capteur.

readSensor

La méthode `readSensor()` transmet la commande spécifiée sur le port SDI-12 au capteur spécifié et renvoie une liste d'objet avec toute les valeurs envoyées par le capteur.

Un exemple réel

Lancez votre interpréteur Python et ouvrez le script correspondant, fourni dans le répertoire **Examples/Doc-GettingStarted-Yocto-SDI12** de la librairie Yoctopuce.

Vous reconnaîtrez dans cet exemple l'utilisation des fonctions expliquées ci-dessus, cette fois utilisées avec le décorum nécessaire à en faire un petit programme d'exemple concret.

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

from yocto_api import *
from yocto_sdi12port import *


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


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


if len(sys.argv) < 2:
    usage()
target = sys.argv[1].upper()

# Setup the API to use local USB devices. You can
# use an IP address instead of 'usb' if the device
# is connected to a network.
errmsg = YRefParam()
if YAPI.RegisterHub("usb", errmsg) != YAPI.SUCCESS:
    sys.exit("init error" + errmsg.value)

if target == 'ANY':
    sdi12Port = YSdi12Port.FirstSdi12Port()
    if sdi12Port is None:
        sys.exit('No module connected (check cable)')
else:
    sdi12Port = YSdi12Port.FirstSdi12Port(sys.argv[1] + ".sdi12port")
    if not sdi12Port.isOnline():
        sys.exit('Module not connected')

singleSensor = sdi12Port.discoverSingleSensor()
print("%-35s %s " % ("Sensor address :", singleSensor.get_sensorAddress()))
print("%-35s %s " % ("Sensor SDI-12 compatibility : " , singleSensor.get_sensorProtocol()))
print("%-35s %s " % ("Sensor company name : " , singleSensor.get_sensorVendor()))
print("%-35s %s " % ("Sensor model number : " , singleSensor.get_sensorModel()))
print("%-35s %s " % ("Sensor version : " , singleSensor.get_sensorVersion()))
print("%-35s %s " % ("Sensor serial number : " , singleSensor.get_sensorSerial()))

valSensor = sdi12Port.readSensor(singleSensor.get_sensorAddress(),"M",5000)
i = 0
```

```
while i < len(valSensor):
    if singleSensor.get_measureCount() > 1:
        print("{0} : {1:8.2f} {2:8s} ({3})".format(singleSensor.get_measureSymbol(i),
                valSensor[i], singleSensor.get_measureUnit(i),
                singleSensor.get_measureDescription(i)))
    else:
        print(valSensor[i])
    i += 1

YAPI.FreeAPI()
```

9.4. Contrôle de la partie module

Chaque module peut-être contrôlé d'une manière similaire, vous trouverez ci-dessous un simple programme d'exemple affichant les principaux paramètres d'un module et permettant d'activer la balise de localisation.

```
#!/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()
```

Chaque propriété `xxx` du module peut être lue grâce à une méthode du type `YModule.get_xxxx()`, et les propriétés qui se sont pas en lecture seule peuvent être modifiées à l'aide de la méthode `YModule.set_xxx()` Pour plus de détails concernant ces fonctions utilisées, reportez-vous aux chapitre API

Modifications des réglages du module

Lorsque que vous souhaitez modifier les réglages d'un module, il suffit d'appeler la fonction `YModule.set_xxx()` correspondante, cependant cette modification n'a lieu que dans la mémoire vive du module: si le module redémarre, les modifications seront perdues. Pour qu'elle soient mémorisées de manière persistante, il est nécessaire de demander au module de sauvegarder sa

configuration courante dans sa mémoire non volatile. Pour cela il faut utiliser la méthode `YModule.saveToFlash()`. Inversement il est possible de forcer le module à oublier ses réglages courants en utilisant la méthode `YModule.revertFromFlash()`. Ce petit exemple ci-dessous vous permet changer le nom logique d'un 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()
```

Attention, le nombre de cycles d'écriture de la mémoire non volatile du module est limité. Passé cette limite plus rien ne garantit que la sauvegarde des réglages se passera correctement. Cette limite, liée à la technologie employée par le micro-processeur du module se situe aux alentour de 100000 cycles. Pour résumer vous ne pouvez employer la fonction `YModule.saveToFlash()` que 100000 fois au cours de la vie du module. Veillez donc à ne pas appeler cette fonction depuis l'intérieur d'une boucle.

Enumeration des modules

Obtenir la liste des modules connectés se fait à l'aide de la fonction `YModule.yFirstModule()` qui renvoie le premier module trouvé, il suffit ensuite d'appeler la mehode `nextModule()` de cet objet pour trouver les modules suivants, et ce tant que la réponse n'est pas un `null`. Ci-dessous un petit exemple listant les module connectés

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


from yocto_api import *

errmsg = YRefParam()

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

print('Device list')

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

9.5. Gestion des erreurs

Lorsque vous implémentez un programme qui doit interagir avec des modules USB, vous ne pouvez pas faire abstraction de la gestion des erreurs. Il y aura forcément une occasion où un utilisateur aura débranché le périphérique, soit avant de lancer le programme, soit même en pleine opération. La librairie Yoctopuce est prévue pour vous aider à supporter ce genre de comportements, mais votre code doit néanmoins être fait pour se comporter au mieux pour interpréter les erreurs signalées par la librairie.

La manière la plus simple de contourner le problème est celle que nous avons employé pour les petits exemples précédents de ce chapitre: avant d'accéder à un module, on vérifie qu'il est en ligne avec la méthode `isOnline()` et on suppose ensuite qu'il va y rester pendant la fraction de seconde nécessaire à exécuter les lignes de code suivantes. Ce n'est pas parfait, mais ça peut suffire dans certains cas. Il faut toutefois être conscient qu'on ne peut pas totalement exclure une erreur se produisant après le `isOnline()`, qui pourrait faire planter le programme. La seule manière de l'éviter est d'implémenter une des deux techniques de gestion des erreurs décrites ci-dessous.

La méthode recommandée par la plupart des langages de programmation pour la gestion des erreurs imprévisibles est l'utilisation d'exceptions. C'est le comportement par défaut de la librairie Yoctopuce. Si une erreur se produit alors qu'on essaie d'accéder à un module, la librairie va lancer une exception. Dans ce cas, de trois choses l'une:

- Si votre code attrape l'exception au vol et la gère, et tout se passe bien.
- Si votre programme tourne dans le debugger, vous pourrez relativement facilement déterminer où le problème s'est produit, et voir le message explicatif lié à l'exception.
- Sinon... l'exception va crasher votre programme, boum!

Comme cette dernière situation n'est pas la plus souhaitable, la librairie Yoctopuce offre une autre alternative pour la gestion des erreurs, permettant de faire un programme robuste sans devoir attraper les exceptions à chaque ligne de code. Il suffit d'appeler la fonction `YAPI.DisableExceptions()` pour commuter la librairie dans un mode où les exceptions de chaque fonction sont systématiquement remplacées par des valeurs de retour particulières, qui peuvent être testées par l'appelant lorsque c'est pertinent. Le nom de la valeur de retour en cas d'erreur pour chaque fonction est systématiquement documenté dans la référence de la librairie. Il suit toujours la même logique: une méthode `get_state()` retournera une valeur `NomDeClasse.STATE_INVALID`, une méthode `get_currentValue` retournera une valeur `NomDeClasse.CURRENTVALUE_INVALID`, etc. Dans tous les cas, la valeur retournée sera du type attendu, et ne sera pas un pointeur nul qui risquerait de faire crasher votre programme. Au pire, si vous affichez la valeur sans la tester, elle sera hors du cadre attendu pour la valeur retournée. Dans le cas de fonctions qui ne retournent à priori pas d'information, la valeur de retour sera `YAPI.SUCCESS` si tout va bien, et un code d'erreur différent en cas d'échec.

Quand vous travaillez sans les exceptions, il est possible d'obtenir un code d'erreur et un message expliquant l'origine de l'erreur en le demandant à l'objet qui a retourné une erreur à l'aide des méthodes `errType()` et `errMessage()`. Ce sont les même informations qui auraient été associées à l'exception si elles avaient été actives.

10. Utilisation du Yocto-SDI12 en C++

Le C++ n'est pas le langage le plus simple à maîtriser. Pourtant, si on prend soin à se limiter aux fonctionnalités essentielles, c'est un langage tout à fait utilisable pour des petits programmes vite faits, et qui a l'avantage d'être très portable d'un système d'exploitation à l'autre. Sous Windows, C++ est supporté avec Microsoft Visual Studio 2017 et les versions plus récentes. Sous macOS, nous supportons les versions de XCode supportées par Apple. Sous Linux, nous supportons toutes les versions de gcc publiées depuis 2008. Par ailleurs, aussi bien sous macOS que sous Linux, vous pouvez compiler les exemples en ligne de commande avec GCC en utilisant le `GNUmakefile` fourni. De même, sous Windows, un `Makefile` vous permet de compiler les exemples en ligne de commande, en pleine connaissance des arguments de compilation et link.

Les librairies Yoctopuce[1] pour C++ vous sont fournies au format source dans leur intégralité. Une partie de la librairie de bas-niveau est écrite en C pur sucre, mais vous n'aurez à priori pas besoin d'interagir directement avec elle: tout a été fait pour que l'interaction soit le plus simple possible depuis le C++. La librairie vous est fournie bien entendu aussi sous forme binaire, de sorte à pouvoir la linker directement si vous le préférez.

Vous allez rapidement vous rendre compte que l'API C++ defini beaucoup de fonctions qui retournent des objets. Vous ne devez jamais désallouer ces objets vous-même. Ils seront désalloués automatiquement par l'API à la fin de l'application.

Afin des les garder simples, tous les exemples fournis dans cette documentation sont des applications consoles. Il va de soit que que les fonctionnement des librairies est strictement identiques si vous les intégrez dans une application dotée d'une interface graphique. Vous trouverez dans la dernière section de ce chapitre toutes les informations nécessaires à la création d'un projet à neuf linké avec les librairies Yoctopuce.

10.1. Contrôle de la fonction Sdi12Port

Il suffit de quelques lignes de code pour piloter un Yocto-SDI12. Voici le squelette d'un fragment de code C++ qui utilise la fonction Sdi12Port.

```
#include "yocto_api.h"
#include "yocto_sdi12port.h"

[...]
// On active la détection des modules sur USB
String errmsg;
YAPI::RegisterHub("usb", errmsg);
```

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

```
[...]

// On récupère l'objet permettant d'intéragir avec le module
YSdi12Port *sdi12port;
sdi12port = YSdi12Port::FindSdi12Port("YSDIMK01-123456.sdi12Port");

// Pour gérer le hot-plug, on vérifie que le module est là
if(sdi12port->isOnline())
{
    // Utiliser sdi12port->set_sdi12Mode()
    [...]
}
```

Voyons maintenant en détail ce que font ces quelques lignes.

yocto_api.h et yocto_sdi12port.h

Ces deux fichiers inclus permettent d'avoir accès aux fonctions permettant de gérer les modules Yoctopuce. `yocto_api.h` doit toujours être utilisé, `yocto_sdi12port.h` est nécessaire pour gérer les modules contenant un port SDI12, comme le Yocto-SDI12.

YAPI::RegisterHub

La fonction `YAPI::RegisterHub` initialise l'API de Yoctopuce en indiquant où les modules doivent être recherchés. Utilisée avec le paramètre `"usb"`, elle permet de travailler avec les modules connectés localement à la machine. Si l'initialisation se passe mal, cette fonction renverra une valeur différente de `YAPI_SUCCESS`, et retournera via le paramètre `errmsg` un explication du problème.

YSdi12Port::FindSdi12Port

La fonction `YSdi12Port::FindSdi12Port` permet de retrouver un port SDI12 en fonction du numéro de série de son module hôte et de son nom de fonction. Mais vous pouvez tout aussi bien utiliser des noms logiques que vous auriez préalablement configurés. Imaginons un module Yocto-SDI12 avec le numéros de série *YSDIMK01-123456* que vous auriez appelé "*MonModule*" et dont vous auriez nommé la fonction *sdi12Port* "*MaFonction*", les cinq appels suivants seront strictement équivalents (pour autant que *MaFonction* ne soit définie qu'une fois, pour éviter toute ambiguïté):

```
YSdi12Port *sdi12port = YSdi12Port::FindSdi12Port("YSDIMK01-123456.sdi12Port");
YSdi12Port *sdi12port = YSdi12Port::FindSdi12Port("YSDIMK01-123456.MaFonction");
YSdi12Port *sdi12port = YSdi12Port::FindSdi12Port("MonModule.sdi12Port");
YSdi12Port *sdi12port = YSdi12Port::FindSdi12Port("MonModule.MaFonction");
YSdi12Port *sdi12port = YSdi12Port::FindSdi12Port("MaFonction");
```

`YSdi12Port::FindSdi12Port` renvoie un objet que vous pouvez ensuite utiliser à loisir pour contrôler le port SDI12.

isOnline

La méthode `isOnline()` de l'objet renvoyé par `YSdi12Port::FindSdi12Port` permet de savoir si le module correspondant est présent et en état de marche.

reset

La méthode `reset()` de l'objet returné par `yFindSerialPort` vide tous les tampons du port série.

discoverSingleSensor

La méthode `discoverSingleSensor()` cherche l'adresse du capteur connecté sur le port SDI-12 et renvoie un objet avec toute les informations du capteur.

readSensor

La méthode `readSensor()` transmet la commande spécifiée sur le port SDI-12 au capteur spécifié et renvoie une liste d'objet avec toute les valeurs envoyées par le capteur.

Un exemple réel

Lancez votre environnement C++ et ouvrez le projet exemple correspondant, fourni dans le répertoire **Examples/Doc-GettingStarted-Yocto-SDI12** de la librairie Yoctopuce. Si vous préférez travailler avec votre éditeur de texte préféré, ouvrez le fichier `main.cpp`, vous taperez simplement `make` dans le répertoire de l'exemple pour le compiler.

Vous reconnaîtrez dans cet exemple l'utilisation des fonctions expliquées ci-dessus, cette fois utilisées avec le décorum nécessaire à en faire un petit programme d'exemple concret.

```
#include "yocto_api.h"
#include "yocto_sdi12port.h"
#include <iostream>
#include <stdlib.h>

using namespace std;

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

int main(int argc, const char * argv[])
{
  string errmsg;
  string target;
  YSdi12Port *sdi12Port;

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

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

  if (target == "any") {
    sdi12Port = YSdi12Port::FirstSdi12Port();
    if (sdi12Port == NULL) {
      cerr << "No module connected (check USB cable)" << endl;
      return 1;
    }
  } else {
    target = (string) argv[1];
    sdi12Port = YSdi12Port::FindSdi12Port(target + ".sdi12Port");
    if (!sdi12Port->isOnline()) {
      cerr << "Module not connected (check USB cable)" << endl;
      return 1;
    }
  }
  sdi12Port->reset();

  YSdi12SensorInfo singleSensor = sdi12Port->discoverSingleSensor();
  printf("%-30s %s \n", "Sensor address :", singleSensor.get_sensorAddress().c_str());
  printf("%-30s %s \n", "Sensor SDI-12 compatibility : ",
         singleSensor.get_sensorProtocol().c_str());
  printf("%-30s %s \n", "Sensor company name : ", singleSensor.get_sensorVendor().c_str());
  printf("%-30s %s \n", "Sensor model number : ", singleSensor.get_sensorModel().c_str());
  printf("%-30s %s \n", "Sensor version : ", singleSensor.get_sensorVersion().c_str());
  printf("%-30s %s \n", "Sensor serial number : ", singleSensor.get_sensorSerial().c_str()
);

  // of the Yocto-SDI12 as well if used
  vector<double> valSensor = sdi12Port->readSensor(singleSensor.get_sensorAddress(), "M",
                                                  5000);
  for (unsigned i = 0; i < valSensor.size(); i++) {
```

```
    if (singleSensor.get_measureCount() > 1) {
      printf("%s %-8.2f%-8s (%s) \n", singleSensor.get_measureSymbol(i).c_str(), valSensor
[i],
             singleSensor.get_measureUnit(i).c_str(), singleSensor.get_measureDescription(i
).c_str());
    } else {
      printf("%.2f \n", valSensor[i]);
    }

  }

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

10.2. Contrôle de la partie module

Chaque module peut-être contrôlé d'une manière similaire, vous trouverez ci dessous un simple programme d'exemple affichant les principaux paramètres d'un module et permettant d'activer la balise de localisation.

```
#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;
}
```

Chaque propriété `xxx` du module peut être lue grâce à une méthode du type `get_xxxx()`, et les propriétés qui se sont pas en lecture seule peuvent être modifiées à l'aide de la méthode `set_xxx()` Pour plus de détails concernant ces fonctions utilisées, reportez-vous aux chapitre API

Modifications des réglages du module

Lorsque que vous souhaitez modifier les réglages d'un module, il suffit d'appeler la fonction `set_xxx()` correspondante, cependant cette modification n'a lieu que dans la mémoire vive du module: si le module redémarre, les modifications seront perdues. Pour qu'elle soient mémorisées de manière persistante, il est nécessaire de demander au module de sauvegarder sa configuration courante dans sa mémoire non volatile. Pour cela il faut utiliser la méthode `saveToFlash()`. Inversement il est possible de forcer le module à oublier ses réglages courants en utilisant la méthode `revertFromFlash()`. Ce petit exemple ci-dessous vous permet changer le nom logique d'un 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;
}
```

Attention, le nombre de cycles d'écriture de la mémoire non volatile du module est limité. Passé cette limite plus rien ne garantit que la sauvegarde des réglages se passera correctement. Cette limite, liée à la technologie employée par le micro-processeur du module se situe aux alentour de 100000 cycles. Pour résumer vous ne pouvez employer la fonction `saveToFlash()` que 100000 fois au cours de la vie du module. Veillez donc à ne pas appeler cette fonction depuis l'intérieur d'une boucle.

Enumeration des modules

Obtenir la liste des modules connectés se fait à l'aide de la fonction `yFirstModule()` qui renvoie le premier module trouvé, il suffit ensuite d'appeler la fonction `nextModule()` de cet objet pour trouver les modules suivants, et ce tant que la réponse n'est pas un `NULL`. Ci-dessous un petit exemple listant les module connectés

```
#include <iostream>

#include "yocto_api.h"

using namespace std;

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

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

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

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

10.3. Gestion des erreurs

Lorsque vous implémentez un programme qui doit interagir avec des modules USB, vous ne pouvez pas faire abstraction de la gestion des erreurs. Il y aura forcément une occasion où un utilisateur aura débranché le périphérique, soit avant de lancer le programme, soit même en pleine opération. La librairie Yoctopuce est prévue pour vous aider à supporter ce genre de comportements, mais votre code doit néanmoins être fait pour se comporter au mieux pour interpréter les erreurs signalées par la librairie.

La manière la plus simple de contourner le problème est celle que nous avons employé pour les petits exemples précédents de ce chapitre: avant d'accéder à un module, on vérifie qu'il est en ligne avec la méthode `isOnline()` et on suppose ensuite qu'il va y rester pendant la fraction de seconde nécessaire à exécuter les lignes de code suivantes. Ce n'est pas parfait, mais ça peut suffire dans certains cas. Il faut toutefois être conscient qu'on ne peut pas totalement exclure une erreur se produisant après le `isOnline()`, qui pourrait faire planter le programme. La seule manière de l'éviter est d'implémenter une des deux techniques de gestion des erreurs décrites ci-dessous.

La méthode recommandée par la plupart des langages de programmation pour la gestion des erreurs imprévisibles est l'utilisation d'exceptions. C'est le comportement par défaut de la librairie Yoctopuce. Si une erreur se produit alors qu'on essaie d'accéder à un module, la librairie va lancer une exception. Dans ce cas, de trois choses l'une:

- Si votre code attrape l'exception au vol et la gère, et tout se passe bien.
- Si votre programme tourne dans le debugger, vous pourrez relativement facilement déterminer où le problème s'est produit, et voir le message explicatif lié à l'exception.
- Sinon... l'exception va crasher votre programme, boum!

Comme cette dernière situation n'est pas la plus souhaitable, la librairie Yoctopuce offre une autre alternative pour la gestion des erreurs, permettant de faire un programme robuste sans devoir

attraper les exceptions à chaque ligne de code. Il suffit d'appeler la fonction YAPI.DisableExceptions() pour commuter la librairie dans un mode où les exceptions de chaque fonction sont systématiquement remplacées par des valeurs de retour particulières, qui peuvent être testées par l'appelant lorsque c'est pertinent. Le nom de la valeur de retour en cas d'erreur pour chaque fonction est systématiquement documenté dans la référence de la librairie. Il suit toujours la même logique: une méthode get_state() retournera une valeur NomDeClasse.STATE_INVALID, une méthode get_currentValue retournera une valeur NomDeClasse.CURRENTVALUE_INVALID, etc. Dans tous les cas, la valeur retournée sera du type attendu, et ne sera pas un pointeur nul qui risquerait de faire crasher votre programme. Au pire, si vous affichez la valeur sans la tester, elle sera hors du cadre attendu pour la valeur retournée. Dans le cas de fonctions qui ne retournent à priori pas d'information, la valeur de retour sera YAPI.SUCCESS si tout va bien, et un code d'erreur différent en cas d'échec.

Quand vous travaillez sans les exceptions, il est possible d'obtenir un code d'erreur et un message expliquant l'origine de l'erreur en le demandant à l'objet qui a retourné une erreur à l'aide des méthodes errType() et errMessage(). Ce sont les même informations qui auraient été associées à l'exception si elles avaient été actives.

10.4. Intégration de la librairie Yoctopuce en C++

Selon vos besoins et vos préférences, vous pouvez être mené à intégrer de différentes manières la librairie à vos projets. Cette section explique comment implémenter les différentes options.

Intégration au format source (recommandé)

L'intégration de toutes les sources de la librairie dans vos projets a plusieurs avantages:

- Elle garanti le respect des conventions de compilation de votre projet (32/64 bits, inclusion des symboles de debug, caractères unicode ou ASCII, etc.);
- Elle facilite le déboggage si vous cherchez la cause d'un problème lié à la librairie Yoctopuce
- Elle réduit les dépendances sur des composants tiers, par exemple pour parer au cas où vous pourriez être mené à recompiler ce projet pour une architecture différente dans de nombreuses années.
- Elle ne requiert pas l'installation d'une librairie dynamique spécifique à Yoctopuce sur le système final, tout est dans l'exécutable.

Pour intégrer le code source, le plus simple est d'inclure simplement le répertoire Sources de la librairie Yoctopuce à votre **IncludePath**, et d'ajouter tous les fichiers de ce répertoire (y compris le sous-répertoire yapi) à votre projet.

Pour que votre projet se construise ensuite correctement, il faudra linker avec votre projet les librairies systèmes requises, à savoir:

- Pour Windows: les librairies sont mises automatiquement
- Pour macOS: **IOKit.framework** et **CoreFoundation.framework**
- Pour Linux: **libm**, **libpthread**, **libusb1.0** et **libstdc++**

Intégration en librairie statique

L'intégration de de la librairie Yoctopuce sous forme de librairie statique permet une compilation rapide du programme en une seule commande. Elle ne requiert pas non plus l'installation d'une librairie dynamique spécifique à Yoctopuce sur le système final, tout est dans l'exécutable.

Pour utiliser la librairie statique, il faut la compiler à l'aide du shell script build.sh sous UNIX, ou build.bat sous Windows. Ce script qui se situe à la racine de la librairie, détecte l'OS et recompile toutes les librairies ainsi que les exemples correspondants.

Ensuite, pour intégrer la librairie statique Yoctopuce à votre projet, vous devez inclure le répertoire Sources de la librairie Yoctopuce à votre **IncludePath**, et ajouter le sous-répertoire de Binaries/... correspondant à votre système d'exploitation à votre **LibPath**.

Finalement, pour que votre projet se construise ensuite correctement, il faudra linker avec votre projet la librairie Yoctopuce et les librairies systèmes requises:

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

Attention, sous Linux, si vous voulez compiler en ligne de commande avec GCC, il est en général souhaitable de linker les librairies systèmes en dynamique et non en statique. Pour mélanger sur la même ligne de commande des librairies statiques et dynamiques, il faut passer les arguments suivants:

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

Intégration en librairie dynamique

L'intégration de la librairie Yoctopuce sous forme de librairie dynamique permet de produire un exécutable plus petit que les deux méthodes précédentes, et de mettre éventuellement à jour cette librairie si un correctif s'avérait nécessaire sans devoir recompiler le code source de l'application. Par contre, c'est un mode d'intégration qui exigera systématiquement de copier la librairie dynamique sur la machine cible ou l'application devra être lancée (**yocto.dll** sous Windows, **libyocto.so.1.0.1** sous macOS et Linux).

Pour utiliser la librairie dynamique, il faut la compiler à l'aide du shell script `build.sh` sous UNIX, ou `build.bat` sous Windows. Ce script qui se situe à la racine de la librairie, détecte l'OS et recompile toutes les librairies ainsi que les exemples correspondant.

Ensuite, pour intégrer la librairie dynamique Yoctopuce à votre projet, vous devez inclure le répertoire `Sources` de la librairie Yoctopuce à votre **IncludePath**, et ajouter le sous-répertoire de `Binaries/...` correspondant à votre système d'exploitation à votre **LibPath**.

Finalement, pour que votre projet se construise ensuite correctement, il faudra linker avec votre projet la librairie dynamique Yoctopuce et les librairies systèmes requises:

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

Avec GCC, la ligne de commande de compilation est simplement:

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

11. Utilisation du Yocto-SDI12 en C#

C# (prononcez C-Sharp) est un langage orienté objet promu par Microsoft qui n'est pas sans rappeller Java. Tout comme Visual Basic et Delphi, il permet de créer des applications Windows relativement facilement. C# est supporté sous Windows Visual Studio 2017 et ses versions plus récentes.

Notre librairie est aussi compatible avec *Mono*, la version open source de C# qui fonctionne sous Linux et macOS. Sous Linux, utilisez la version 5.20 ou plus récente. Le support de Mono sous macOS est limité aux systèmes 32bits, ce qui le rend de nos jours à peu près inutile.Vous trouverez sur notre site web différents articles qui décrivent comment indiquer à Mono comment accéder à notre librairie.

11.1. Installation

Téléchargez la librairie Yoctopuce pour Visual C# depuis le site web de Yoctopuce[1]. Il n'y a pas de programme d'installation, copiez simplement de contenu du fichier zip dans le répertoire de votre choix. Vous avez besoin essentiellement du contenu du répertoire `Sources`. Les autres répertoires contiennent la documentation et quelques programmes d'exemple. Les projets d'exemple sont des projets Visual C# 2010, si vous utilisez une version antérieure, il est possible que vous ayez à reconstruire la structure de ces projets.

11.2. Utilisation l'API yoctopuce dans un projet Visual C#

La librairie Yoctopuce pour Visual C# .NET se présente sous la forme d'une DLL et de fichiers sources en Visual C#. La DLL n'est pas une DLL .NET mais une DLL classique, écrite en C, qui gère les communications à bas niveau avec les modules[2]. Les fichiers sources en Visual C# gèrent la partie haut niveau de l'API. Vous avez donc besoin de cette DLL et des fichiers .cs du répertoire `Sources` pour créer un projet gérant des modules Yoctopuce.

Configuration d'un projet Visual C#

Les indications ci-dessous sont fournies pour Visual Studio express 2010, mais la procédure est semblable pour les autres versions.

Commencez par créer votre projet, puis depuis le panneau **Explorateur de solutions** effectuez un clic droit sur votre projet, et choisissez **Ajouter** puis **Elément existant**.

[1] www.yoctopuce.com/FR/libraries.php
[2] Les sources de cette DLL sont disponibles dans l'API C++

Une fenêtre de sélection de fichiers apparaît: sélectionnez le fichier `yocto_api.cs` et les fichiers correspondant aux fonctions des modules Yoctopuce que votre projet va gérer. Dans le doute, vous pouvez aussi sélectionner tous les fichiers.

Vous avez alors le choix entre simplement ajouter ces fichiers à votre projet, ou les ajouter en tant que lien (le bouton **Ajouter** est en fait un menu déroulant). Dans le premier cas, Visual Studio va copier les fichiers choisis dans votre projet, dans le second Visual Studio va simplement garder un lien sur les fichiers originaux. Il est recommandé d'utiliser des liens, une éventuelle mise à jour de la librairie sera ainsi beaucoup plus facile.

Ensuite, ajoutez de la même manière la dll `yapi.dll`, qui se trouve dans le répertoire `Sources/dll`[3]. Puis depuis la fenêtre **Explorateur de solutions**, effectuez un clic droit sur la DLL, choisissez **Propriété** et dans le panneau **Propriétés**, mettez l'option **Copier dans le répertoire de sortie** à **toujours copier**. Vous êtes maintenant prêt à utiliser vos modules Yoctopuce depuis votre environnement Visual Studio.

Afin de les garder simples, tous les exemples fournis dans cette documentation sont des applications consoles. Il va de soit que que les fonctionnement des librairies est strictement identiques si vous les intégrez dans une application dotée d'une interface graphique.

11.3. Contrôle de la fonction Sdi12Port

Il suffit de quelques lignes de code pour piloter un Yocto-SDI12. Voici le squelette d'un fragment de code C# qui utilise la fonction Sdi12Port.

```
[...]
// On active la détection des modules sur USB
string errmsg = "";
YAPI.RegisterHub("usb", errmsg);
[...]

// On récupère l'objet permettant d'intéragir avec le module
YSdi12Port sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.sdi12Port");

// Pour gérer le hot-plug, on vérifie que le module est là
if (sdi12port.isOnline())
{
    // Utiliser sdi12port.set_sdi12Mode()
    [...]
}
```

Voyons maintenant en détail ce que font ces quelques lignes.

YAPI.RegisterHub

La fonction `YAPI.RegisterHub` initialise l'API de Yoctopuce en indiquant où les modules doivent être recherchés. Utilisée avec le paramètre `"usb"`, elle permet de travailler avec les modules connectés localement à la machine. Si l'initialisation se passe mal, cette fonction renverra une valeur différente de `YAPI.SUCCESS`, et retournera via le paramètre `errmsg` une explication du problème.

YSdi12Port.FindSdi12Port

La fonction `YSdi12Port.FindSdi12Port` permet de retrouver un port SDI12 en fonction du numéro de série de son module hôte et de son nom de fonction. Mais vous pouvez tout aussi bien utiliser des noms logiques que vous auriez préalablement configurés. Imaginons un module Yocto-SDI12 avec le numéros de série *YSDIMK01-123456* que vous auriez appelé "*MonModule*" et dont vous auriez nommé la fonction *sdi12Port* "*MaFonction*", les cinq appels suivants seront strictement équivalents (pour autant que *MaFonction* ne soit définie qu'une fois, pour éviter toute ambiguïté):

```
sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.sdi12Port");
sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.MaFonction");
sdi12port = YSdi12Port.FindSdi12Port("MonModule.sdi12Port");
```

[3] Pensez à changer le filtre de la fenêtre de sélection de fichiers, sinon la DLL n'apparaîtra pas

```
sdi12port = YSdi12Port.FindSdi12Port("MonModule.MaFonction");
sdi12port = YSdi12Port.FindSdi12Port("MaFonction");
```

`YSdi12Port.FindSdi12Port` renvoie un objet que vous pouvez ensuite utiliser à loisir pour contrôler le port SDI12.

isOnline

La méthode `isOnline()` de l'objet renvoyé par `YSdi12Port.FindSdi12Port` permet de savoir si le module correspondant est présent et en état de marche.

reset

La méthode `reset()` de l'objet returné par `YSdi12Port.FindSerialPort` vide tous les tampons du port série.

discoverSingleSensor

La méthode `discoverSingleSensor()` cherche l'adresse du capteur connecté sur le port SDI-12 et renvoie un objet avec toute les informations du capteur.

readSensor

La méthode `readSensor()` transmet la commande spécifiée sur le port SDI-12 au capteur spécifié et renvoie une liste d'objet avec toute les valeurs envoyées par le capteur.

Un exemple réel

Lancez Visual C# et ouvrez le projet exemple correspondant, fourni dans le répertoire **Examples/ Doc-GettingStarted-Yocto-SDI12** de la librairie Yoctopuce.

Vous reconnaîtrez dans cet exemple l'utilisation des fonctions expliquées ci-dessus, cette fois utilisées avec le décorum nécessaire à en faire un petit programme d'exemple concret.

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

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

    static void Main(string[] args)
    {
      string errmsg = "";
      string target;
      YSdi12Port sdi12Port;

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

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

      if (target == "ANY") {
        sdi12Port = YSdi12Port.FirstSdi12Port();
        if (sdi12Port == null) {
```

```
          Console.WriteLine("No module connected (check USB cable) ");
          Environment.Exit(0);
        }
        target = sdi12Port.get_module().get_serialNumber();
      }

      sdi12Port = YSdi12Port.FindSdi12Port(target + ".sdi12Port");
      if (sdi12Port.isOnline()) {
        sdi12Port.reset();
        YSdi12SensorInfo singleSensor = sdi12Port.discoverSingleSensor();
        Console.WriteLine("Sensor address : " + singleSensor.get_sensorAddress());
        Console.WriteLine("Sensor SDI-12 compatibility : " +
singleSensor.get_sensorProtocol());
        Console.WriteLine("Sensor company name : " + singleSensor.get_sensorVendor());
        Console.WriteLine("Sensor model number : " + singleSensor.get_sensorModel());
        Console.WriteLine("Sensor version : " + singleSensor.get_sensorVersion());
        Console.WriteLine("Sensor serial number : " + singleSensor.get_sensorSerial());

        List<double> valSensor = sdi12Port.readSensor(singleSensor.get_sensorAddress(),
"M",
                                5000);
        Console.WriteLine("Sensor: " + singleSensor.get_sensorAddress());

        for (int i = 0; i < valSensor.Count; i++) {
          if (singleSensor.get_measureCount() > 1) {
            Console.WriteLine(String.Format("{0} : {1,-8:0.00} {2,-10} ({3})",
                                            singleSensor.get_measureSymbol(i), valSensor[i
], singleSensor.get_measureUnit(i),
                                            singleSensor.get_measureDescription(i)));
          } else {
            Console.WriteLine(valSensor[i]);
          }
        }
      }

      YAPI.FreeAPI();

      // wait 5 sec to show the output
      ulong now = YAPI.GetTickCount();
      while (YAPI.GetTickCount() - now < 5000);
    }
  }
}
```

11.4. Contrôle de la partie module

Chaque module peut-être contrôlé d'une manière similaire, vous trouverez ci-dessous un simple programme d'exemple affichant les principaux paramètres d'un module et permettant d'activer la balise de localisation.

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


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

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

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


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

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

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

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

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

Chaque propriété `xxx` du module peut être lue grâce à une méthode du type `YModule.get_xxxx()`, et les propriétés qui se sont pas en lecture seule peuvent être modifiées à l'aide de la méthode `YModule.set_xxx()` Pour plus de détails concernant ces fonctions utilisées, reportez-vous aux chapitre API

Modifications des réglages du module

Lorsque que vous souhaitez modifier les réglages d'un module, il suffit d'appeler la fonction `YModule.set_xxx()` correspondante, cependant cette modification n'a lieu que dans la mémoire vive du module: si le module redémarre, les modifications seront perdues. Pour qu'elle soient mémorisées de manière persistante, il est nécessaire de demander au module de sauvegarder sa configuration courante dans sa mémoire non volatile. Pour cela il faut utiliser la méthode `YModule.saveToFlash()`. Inversement il est possible de forcer le module à oublier ses réglages courants en utilisant la méthode `YModule.revertFromFlash()`. Ce petit exemple ci-dessous vous permet changer le nom logique d'un 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();
    }
  }
}
```

Attention, le nombre de cycles d'écriture de la mémoire non volatile du module est limité. Passé cette limite plus rien ne garantit que la sauvegarde des réglages se passera correctement. Cette limite, liée à la technologie employée par le micro-processeur du module se situe aux alentour de 100000 cycles. Pour résumer vous ne pouvez employer la fonction `YModule.saveToFlash()` que 100000 fois au cours de la vie du module. Veillez donc à ne pas appeler cette fonction depuis l'intérieur d'une boucle.

Enumeration des modules

Obtenir la liste des modules connectés se fait à l'aide de la fonction `YModule.yFirstModule()` qui renvoie le premier module trouvé, il suffit ensuite d'appeler la méthode `nextModule()` de cet objet pour trouver les modules suivants, et ce tant que la réponse n'est pas un `null`. Ci-dessous un petit exemple listant les module connectés

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

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

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

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

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

11.5. Gestion des erreurs

Lorsque vous implémentez un programme qui doit interagir avec des modules USB, vous ne pouvez pas faire abstraction de la gestion des erreurs. Il y aura forcément une occasion où un utilisateur aura débranché le périphérique, soit avant de lancer le programme, soit même en pleine opération. La librairie Yoctopuce est prévue pour vous aider à supporter ce genre de comportements, mais votre code doit néanmoins être fait pour se comporter au mieux pour interpréter les erreurs signalées par la librairie.

La manière la plus simple de contourner le problème est celle que nous avons employé pour les petits exemples précédents de ce chapitre: avant d'accéder à un module, on vérifie qu'il est en ligne avec la méthode `isOnline()` et on suppose ensuite qu'il va y rester pendant la fraction de seconde nécessaire à exécuter les lignes de code suivantes. Ce n'est pas parfait, mais ça peut suffire dans certains cas. Il faut toutefois être conscient qu'on ne peut pas totalement exclure une erreur se produisant après le `isOnline()`, qui pourrait faire planter le programme. La seule manière de l'éviter est d'implémenter une des deux techniques de gestion des erreurs décrites ci-dessous.

La méthode recommandée par la plupart des langages de programmation pour la gestion des erreurs imprévisibles est l'utilisation d'exceptions. C'est le comportement par défaut de la librairie Yoctopuce. Si une erreur se produit alors qu'on essaie d'accéder à un module, la librairie va lancer une exception. Dans ce cas, de trois choses l'une:

- Si votre code attrape l'exception au vol et la gère, et tout se passe bien.
- Si votre programme tourne dans le debugger, vous pourrez relativement facilement déterminer où le problème s'est produit, et voir le message explicatif lié à l'exception.
- Sinon... l'exception va crasher votre programme, boum!

Comme cette dernière situation n'est pas la plus souhaitable, la librairie Yoctopuce offre une autre alternative pour la gestion des erreurs, permettant de faire un programme robuste sans devoir attraper les exceptions à chaque ligne de code. Il suffit d'appeler la fonction `YAPI.DisableExceptions()` pour commuter la librairie dans un mode où les exceptions de chaque fonction sont systématiquement remplacées par des valeurs de retour particulières, qui peuvent être testées par l'appelant lorsque c'est pertinent. Le nom de la valeur de retour en cas d'erreur pour chaque fonction est systématiquement documenté dans la référence de la librairie. Il suit toujours la même logique: une méthode `get_state()` retournera une valeur `NomDeClasse.STATE_INVALID`, une méthode `get_currentValue` retournera une valeur `NomDeClasse.CURRENTVALUE_INVALID`, etc. Dans tous les cas, la valeur retournée sera du type attendu, et ne sera pas un pointeur nul qui risquerait de faire crasher votre programme. Au pire, si vous affichez la valeur sans la tester, elle sera hors du cadre attendu pour la valeur retournée. Dans le cas de fonctions qui ne retournent à priori pas d'information, la valeur de retour sera `YAPI.SUCCESS` si tout va bien, et un code d'erreur différent en cas d'échec.

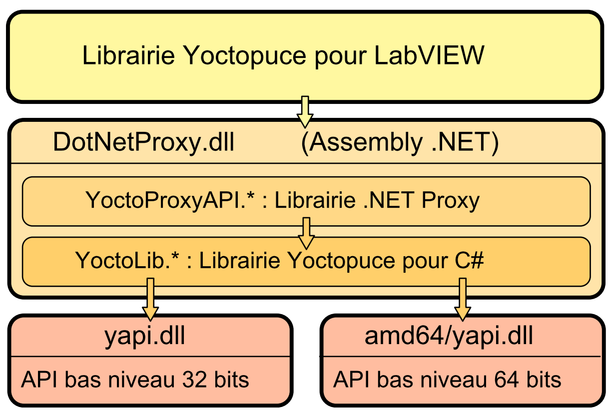
Quand vous travaillez sans les exceptions, il est possible d'obtenir un code d'erreur et un message expliquant l'origine de l'erreur en le demandant à l'objet qui a retourné une erreur à l'aide des méthodes `errType()` et `errMessage()`. Ce sont les même informations qui auraient été associées à l'exception si elles avaient été actives.

12. Utilisation du Yocto-SDI12 avec LabVIEW

LabVIEW est édité par National Instruments depuis 1986. C'est un environnement de développement graphique: plutôt que d'écrire des lignes de code, l'utilisateur dessine son programme, un peu comme un organigramme. LabVIEW est surtout pensé pour interfacer des instruments de mesures d'où le nom *Virtual Instruments* (VI) des programmes LabVIEW. Avec la programmation visuelle, dessiner des algorithmes complexes devient très vite fastidieux, c'est pourquoi la librairie Yoctopuce pour LabVIEW a été pensée pour être aussi simple de possible à utiliser. Autrement dit, LabVIEW étant un environnement extrêmement différent des autres langages supportés par l'API Yoctopuce, vous rencontrerez des différences majeures entre l'API LabVIEW et les autres API.

12.1. Architecture

La librairie LabVIEW est basée sur la librairie Yoctopuce DotNetProxy contenue dans la DLL DotNetProxyLibrary.dll. C'est en fait cette librairie DotNetProxy qui se charge du gros du travail en s'appuyant sur la librairie Yoctopuce C# qui, elle, utilise l'API bas niveau codée dans yapi.dll (32bits) et amd64\yapi.dll (64bits).



Architecture de l'API Yoctopuce pour LabVIEW

Vos applications LabVIEW utilisant l'API Yoctopuce devront donc impérativement être distribuées avec les DLL *DotNetProxyLibrary.dll*, *yapi.dll* et *amd64\yapi.dll*

Si besoin est, vous trouverez les sources de l'API bas niveau dans la librairie C# et les sources de *DotNetProxyLibrary.dll* dans la librairie *DotNetProxy*.

12.2. Compatibilité

Firmwares

Pour que la librairie Yoctopuce pour LabVIEW fonctionne convenablement avec vos modules Yoctopuce, ces derniers doivent avoir au moins le firmware 37120

LabVIEW pour Linux et MacOS

Au moment de l'écriture de ce manuel, l'API Yoctopuce pour LabVIEW n'a été testée que sous Windows. Il y a donc de fortes chances pour qu'elle ne fonctionne tout simplement pas avec les versions Linux et MacOS de LabVIEW.

LabVIEW NXG

La librairie Yoctopuce pour LabVIEW faisant appel à de nombreuses techniques qui ne sont pas encore disponibles dans la nouvelle génération de LabVIEW, elle n'est absolument pas compatible avec LabVIEW NXG.

A propos de DotNetProxyLibrary.dll

Afin d'être compatible avec un maximum de version de Windows, y compris Windows XP, la librairie *DotNetProxyLibrary.dll* est compilée en .NET 3.5, qui est disponible par défaut sur toutes les versions de Windows depuis XP.

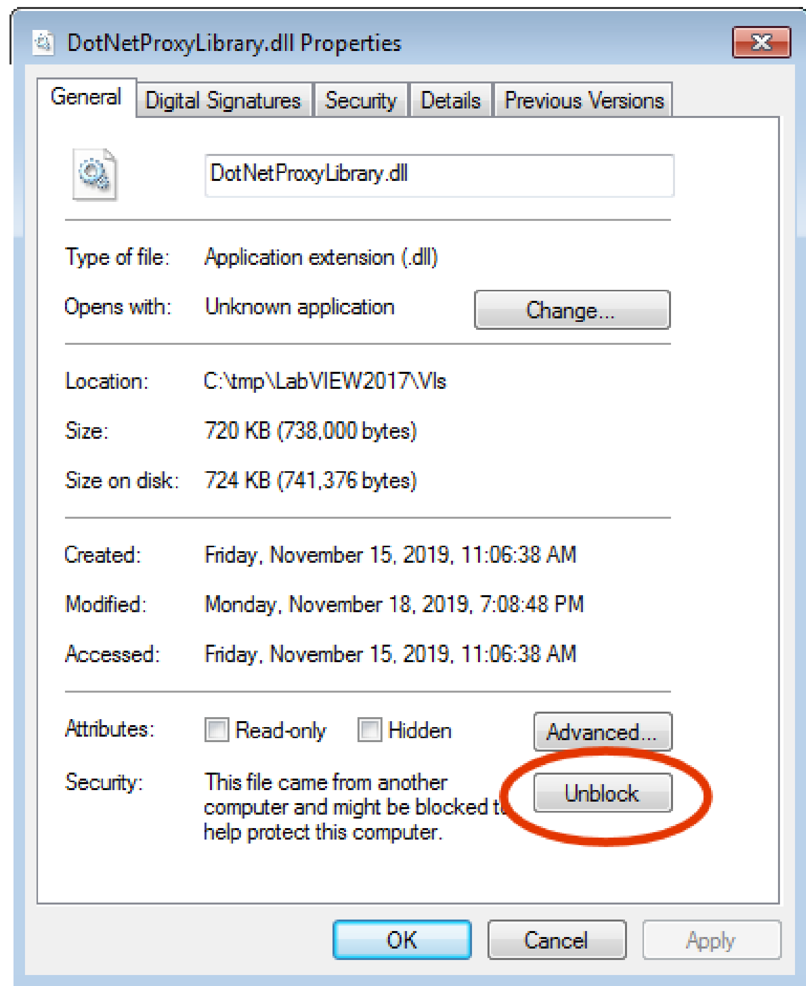
12.3. Installation

Téléchargez la librairie pour LabVIEW depuis le site web de Yoctopuce[1]. Il s'agit d'un fichier ZIP dans lequel vous trouverez un répertoire par version de LabVIEW. Chacun de ses répertoires contient deux sous-répertoires. Le premier contient des exemples de programmation pour chaque produit Yoctopuce; le second, nommé *VIs*, contient tous les VI de l'API et les DLL nécessaires.

Suivant la configuration de Windows et la méthode utilisée pour la copier, la DLL *DotNetProxyLibrary.dll* peut se faire bloquer par Windows parce que ce dernier aura détecté qu'elle provient d'une autre machine. Un cas typique est la décompression de l'archive de la librairie avec l'explorateur de fichier de Windows. Si la DLL est bloquée, LabVIEW ne pourra pas la charger, ce qui entrainera une erreur 1386 lors de l'exécution de n'importe quel VI de la librairie Yoctopuce.

Il y a deux manières de corriger le problème. La plus simple consiste à utiliser l'explorateur de fichier de Windows pour afficher les propriétés de la DLL et la débloquer. Mais cette manipulation devra être répété à chaque fois qu'une nouvelle version de la DLL sera copiée sur votre système.

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



Débloquer la DLL DotNetProxyLibrary.dll.

La seconde méthode consiste à créer dans le même répertoire que l'exécutable labview.exe un fichier XML nommé *labview.exe.config* et contenant le code suivant :

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

Veillez à choisir le bon répertoire en fonction de la version de LabVIEW que vous utilisez (32 bits vs 64 bits). Vous trouverez plus d'information à propos de ce fichier sur le site web de National Instrument[2].

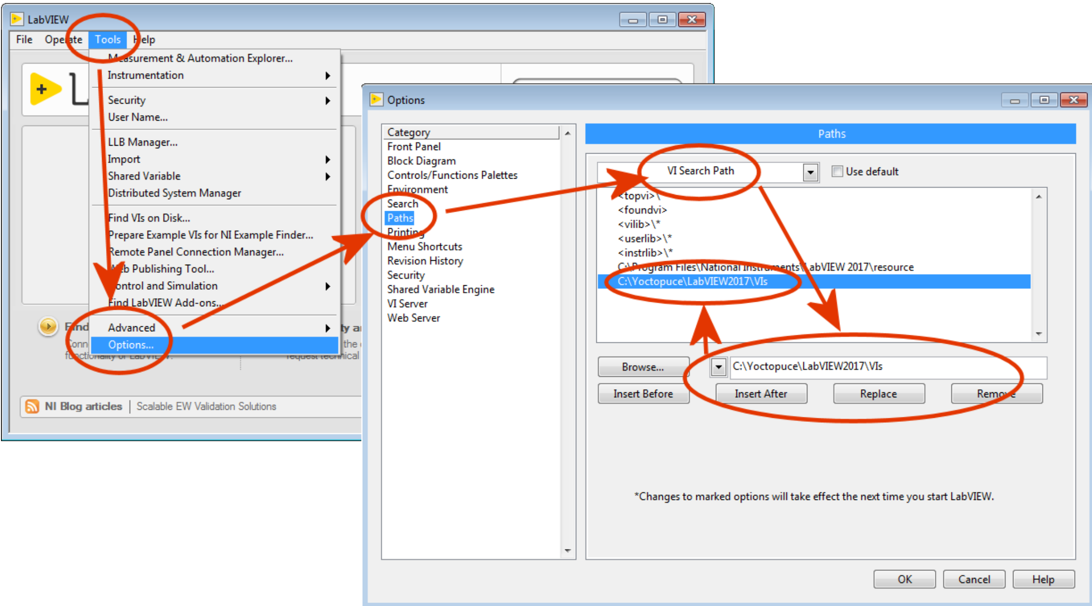
Pour installer l'API Yoctopuce pour LabVIEW vous avez plusieurs méthodes à votre disposition.

Méthode 1 : Installation "à l'emporter"

La manière la plus simple pour installer la librairie Yoctopuce consiste à copier le contenu du répertoire *VIs* où bon vous semble, et à utiliser les VIs dans LabVIEW avec une simple opération de *Drag and Drop*.

Pour pouvoir utiliser les exemples fournis avec l'API, vous aurez avantage à ajouter le répertoire des VIs Yoctopuce dans la liste des répertoires où LabVIEW doit chercher les VIs qu'il n'a pas trouvé. Cette liste est accessible via le menu *Tools* > *Options* > *Paths* > *VI Search Path*.

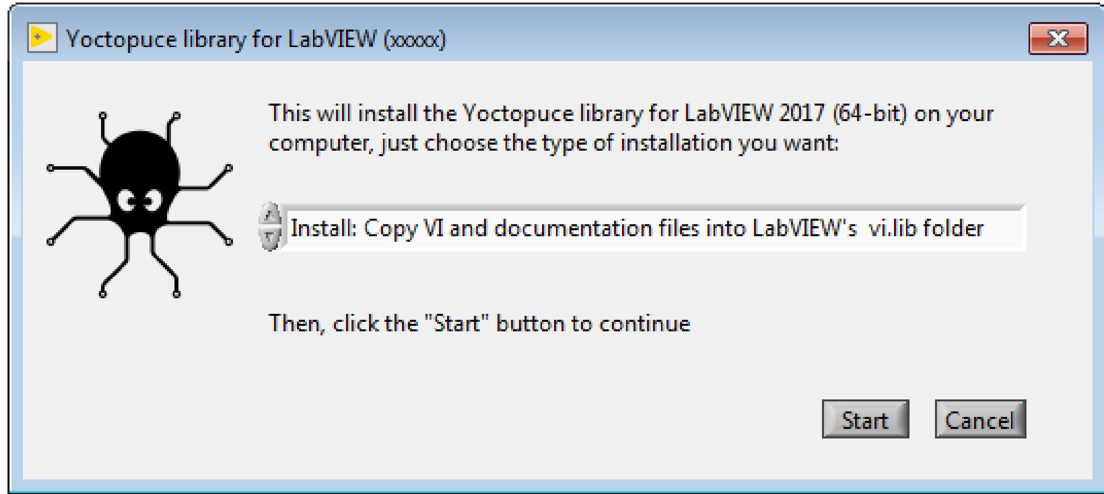
[2] https://knowledge.ni.com/KnowledgeArticleDetails?id=kA00Z000000P8XnSAK



Configuration du "VI Search Path"

Méthode 2 : Installeur fourni avec la librairie

Dans chaque répertoire LabVIEW200xx de la librairie, vous trouverez un VI appelé "*Install.vi*". Ouvrez simplement celui qui correspond à votre version de LabVIEW.



L'installeur fourni avec la librairie

Cet installeur offre trois options d'installation:

Install: Keep VI and documentation files where they are.
Avec cette option, les VI sont conservés à l'endroit où la librairie à été décompressée. Vous aurez donc à faire en sorte qu'ils ne soit pas effacés tant que vous en aurez besoin. Voici ce que fait exactement l'installeur quand cette option est choisie:

- Toute référence à des répertoires contenant une version quelconque de la librairie Yoctopuce sont supprimés de l'option *viSearchPath* dans le fichier *labview.ini*.
- Un fichier de palette *dir.mnu* référençant les VIs est créé dans le répertoire: *C:\Program Files xx\National Instruments\LabVIEW 20xx\vi.lib\addons\Yoctopuce*
- Une référence au répertoire contenants les VIs sera inséré dans l'option *viSearchPath* du fichier *labview.ini*.

Install: Copy VI and documentation files into LabVIEW's vi.lib folder
Dans ce cas, tous les fichiers nécessaires au bon fonctionnement de la librairie sont copiés dans le répertoire d'installation de LabVIEW. Vous pourrez donc effacer les fichiers originaux une fois

l'installation terminée. Notez cependant que les exemples de programmation ne sont pas copiés. Voici ce que fait l'installeur exactement:

- Toute référence à des répertoires contenant une version quelconque de la librairie Yoctopuce sont supprimés de l'option *viSearchPath* dans le fichier *labview.ini*.
- Tous les VIs, DLL et fichiers d'aide sont copiés dans: *C:\Program Files xx\National Instruments\LabVIEW 20xx\vi.lib\Yoctopuce*
- Les VIs sont modifiés pour que leur aide pointe sur les nouveaux fichiers d'aide.
- un fichier palette *dir.mnu* référençant les VIs copiés sera créé dans le répertoire: *C:\Program Files xx\National Instruments\LabVIEW 20xx\vi.lib\addons\Yoctopuce*

Uninstall Yoctopuce Library
Cette option supprime la Librairie Yoctopuce de votre installation LabVIEW:

- Toute référence à des répertoires contenant une version quelconque de la librairie Yoctopuce sont supprimés de l'option *viSearchPath* dans le fichier *labview.ini*.
- Les répertoires suivants seront supprimé s'ils existent:
  C:\Program Files xx\National Instruments\LabVIEW 20xx\vi.lib\addons\Yoctopuce
  C:\Program Files xx\National Instruments\LabVIEW 20xx\vi.lib\Yoctopuce

Dans tous les cas, si le fichier *labview.ini* a besoin d'être modifié, une copie de backup est automatiquement réalisée avant.

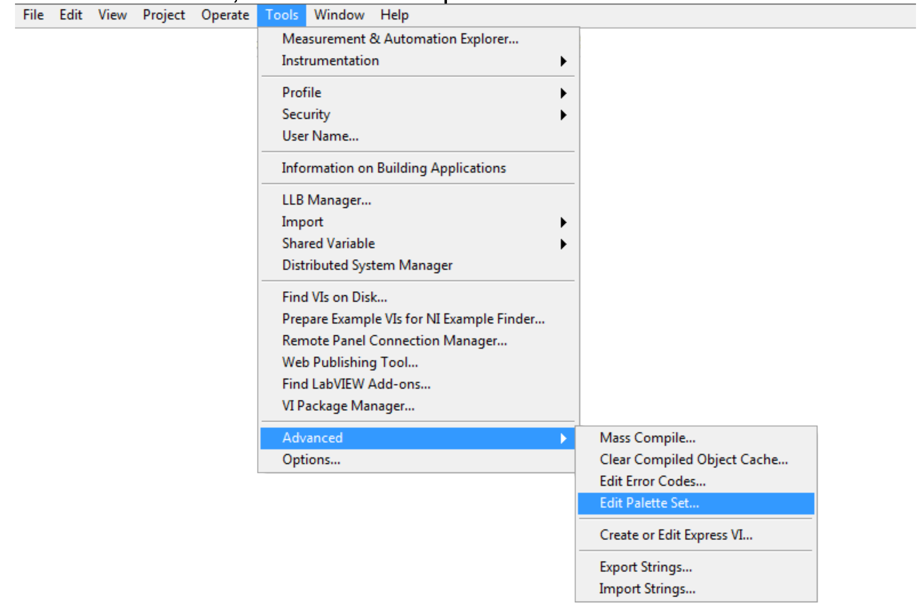
L'installeur reconnait les répertoires contenant la librairie Yoctopuce en testant l'existence du fichier *YRegisterHub.vi*.

Une fois l'installation terminée, vous trouverez une palette Yoctopuce dans le menu *Fonction/ Suppléments*.

Méthode 3 Installation manuelle dans la palette LabVIEW

Les étapes pour installer manuellement les VIs directement dans la Palette LabView sont un peu plus complexes, vous trouverez la procédure complète sur le site de National Instruments[3], mais voici un résumé:

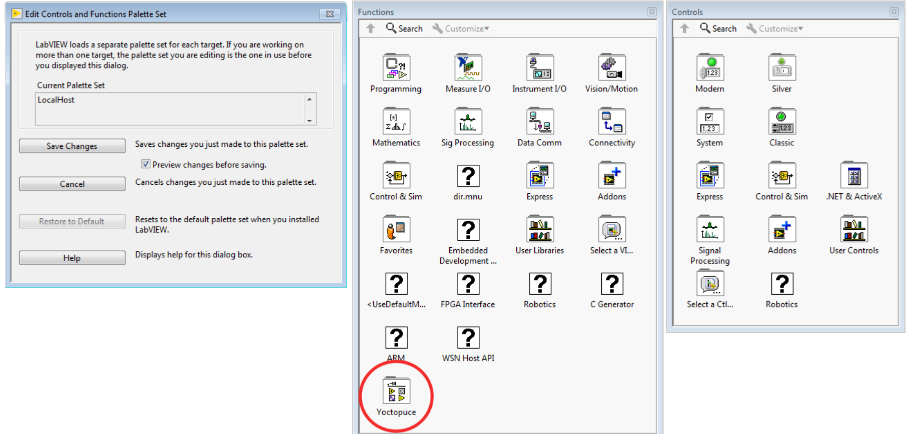
1. Créez un répertoire *Yoctopuce\API* dans le répertoire *C:\Program Files\National Instruments \LabVIEW xxxx\vi.lib*, et copiez tous les VIs et les DLL du répertoire *VIs* dedans.
2. Créez un répertoire *Yoctopuce* dans le répertoire *C:\Program Files\National Instruments \LabVIEW xxxx\menus\Categories*
3. Lancez LabVIEW, et choisissez l'option *Tools>Advanced>Edit Palette Set*



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

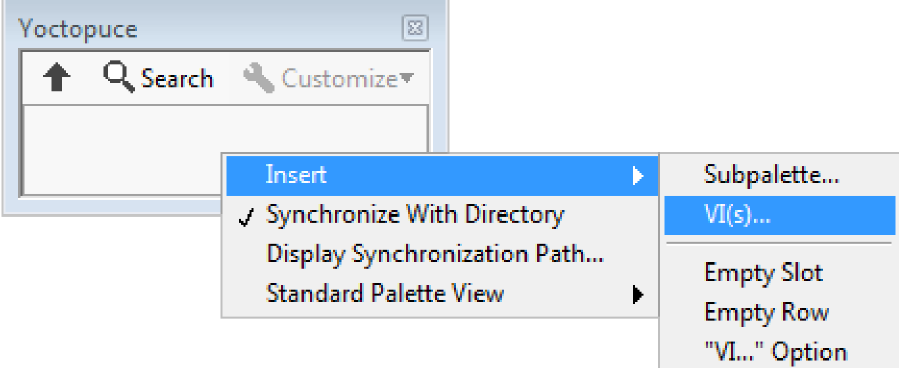
Trois fenêtres vont apparaître:

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

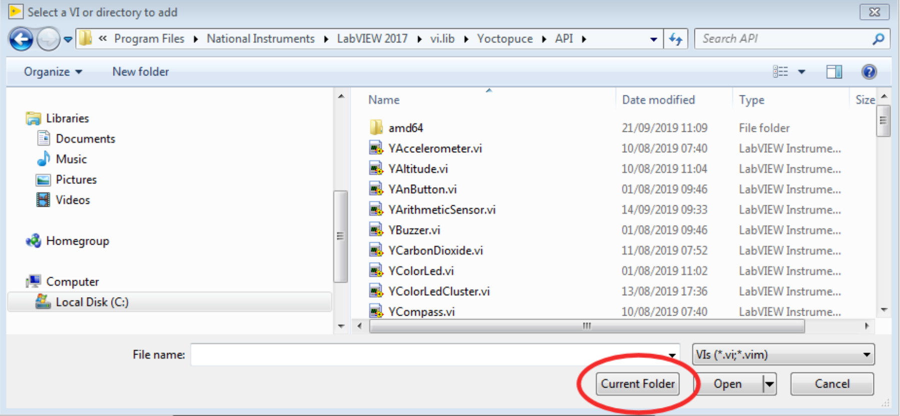


.
Dans la fenêtre *Function*, vous trouverez une icône *Yoctopuce*. Double-cliquez dessus, ce qui fera apparaitre une fenêtre "Yoctopuce" vide.

4. Dans la fenêtre Yoctopuce, faites un *Clic Droit>Insert>Vi(s)..*

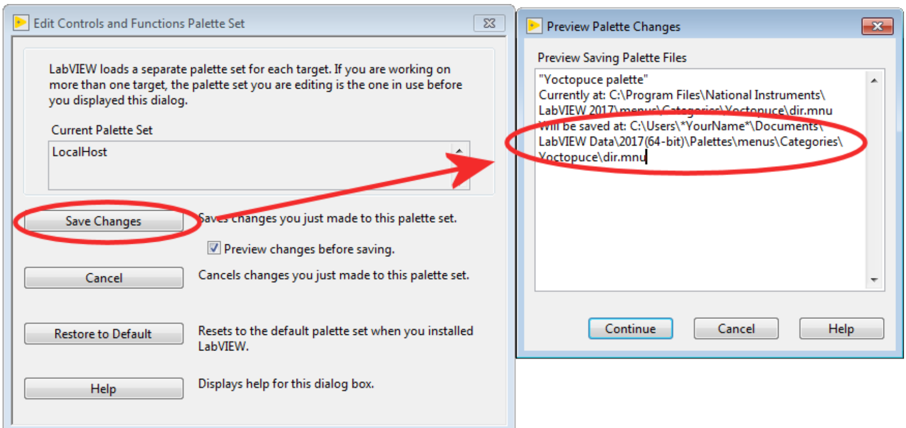


ce qui fera apparaître un sélecteur de fichier. Placer le sélecteur dans le répertoire *vi.lib \Yoctopuce\API* que vous avez créé au point 1 et cliquez sur *Current Folder*



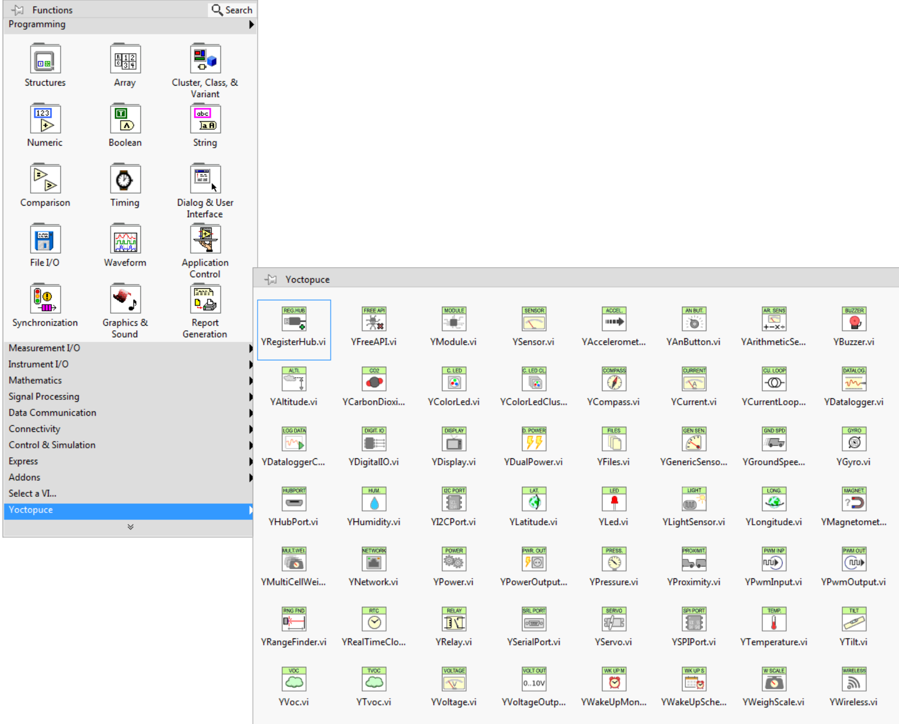
Tous les VIs Yoctopuce vont apparaitre dans la fenêtre Yoctopuce. Par défaut, ils sont triés dans l'ordre alphabétique, mais vous pouvez les arranger comme bon vous semble en les glissant avec la souris. Pour que la palette soit bien utilisable, nous vous suggérons de réorganiser les icônes sur 8 colonnes.

5. Dans la fenêtre "*Edit Controls and Functions Palette Set*", cliquez sur le bouton "Save Changes", la fenêtre va vous indiquer qu'elle a créé un fichier `dir.mnu` dans votre répertoire `Documents`.



Copiez ce fichier dans le répertoire "`menus\Categories\Yoctopuce`" que vous avez créé au point 2.

6. Redémarrez LabVIEW, la palette de LabVIEW contient maintenant une sous-palette Yoctopuce et avec tous les VIs de l'API

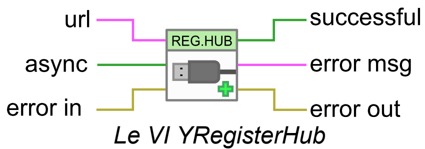


12.4. Présentation des VIs Yoctopuce

La librairie Yoctopuce pour LabVIEW comprend un VI par classe de l'API Yoctopuce, plus quelques VI spéciaux. Tous les VIs disposent des connecteurs traditionnels *Error IN* et *Error Out*.

YRegisterHub

Le VI `YRegisterHub` permet d'initialiser l'API. Ce VI doit impérativement être appelé une fois avant de faire quoi que ce soit qui soit en relation avec des modules Yoctopuce



Le VI YRegisterHub

Le VI `YRegisterHub` prend un paramètre *url* qui peut être soit:

- La chaîne de caractères "`usb`" pour indiquer que l'on souhaite travailler avec des modules locaux directement par USB
- Une addresse IP pour indiquer que l'on souhaite travailler avec des modules accessibles via une connexion réseau. Cette adresse IP peut être celle d'un YoctoHub[4] ou encore celle d'une machine sur laquelle tourne l'application VirtualHub[5].

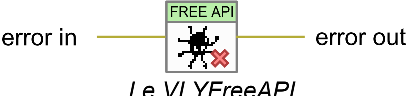
Dans le cas d'une adresse IP, le VI `YRegisterHub` va essayer de contacter cette adresse et génèrera une erreur s'il n'y arrive pas, à moins que le paramètre *async* ne soit mis à TRUE. Si *async* est mis à TRUE, aucune erreur ne sera générée, et les modules Yoctopuce correspondant à cette adresse IP seront automatiquement mis à disposition dès que la machine concernée sera joignable.

Si tout s'est bien passé, la sortie *successful* contiendra la valeur TRUE. Dans le cas contraire elle contiendra la valeur FALSE et la sortie *error msg* contiendra une chaîne de caractères contenant une description de l'erreur

Vous pouvez utiliser plusieurs VI `YRegisterHub` avec des urls différentes si vous le souhaitez. En revanche, sur la même machine, il ne peut y avoir qu'un seul processus qui accède aux modules Yoctopuce locaux directement par USB (*url* mis à "`usb`"). Cette limitation peut facilement être contournée en faisant tourner le logiciel *VirtualHub* sur la machine locale et en utilisant l'url "`127.0.0.1`".

YFreeAPI

Le VI `YFreeAPI` permet de libérer les ressources allouée par l'API Yoctopuce.

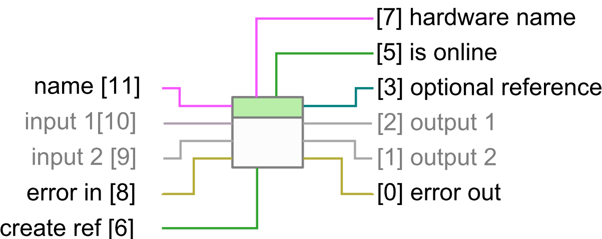


Le VI YFreeAPI

Le VI `YFreeAPI` doit être appelé une fois que votre code en a fini avec l'API Yoctopuce, faute de quoi l'accès direct par USB (*url* mis à "`usb`") pourrait rester bloqué une fois l'exécution de votre VI terminé, et ce tant que LabVIEW n'aura pas été complètement fermé.

Structure des VI correspondant à une classe

Les autres VIs correspondent à une fonction/classe de l'API Yoctopuce, ils ont tous la même structure:



Structure de la plupart des VIs de l'API.

- Connecteur [11]: *name* doit contenir le nom hardware ou le nom logique de la fonction visée.
- Connecteur [10] et [9]: paramètres d'entrée qui dépendent de la nature du VI
- Connecteur [8] et [0] : *error in* et *error out*.
- Connecteur [7] : Nom hardware unique de la fonction trouvée.
- Connecteur [5] : *is online* contient TRUE si la fonction est accessible, FALSE sinon.
- Connecteur [2] et [1]: valeurs de sortie qui dépendent de la nature du VI.
- Connecteur [6]: Si cette entré est mise à TRUE, le connecteur [3] contiendra une référence à l'objet *Proxy* implémenté par le VI[6]. Cette entrée est initialisée à FALSE par défaut.

[4] www.yoctopuce.com/FR/products/category/extensions-and-networking
[5] http://www.yoctopuce.com/EN/virtualhub.php
[6] voir section *Utilisation objets Proxy*

- Connecteur [3]: Référence sur l'objet *Proxy* implémenté par le VI si l'entrée [6] contient TRUE. Cet objet permet d'accéder à des fonctionnalités supplémentaires.

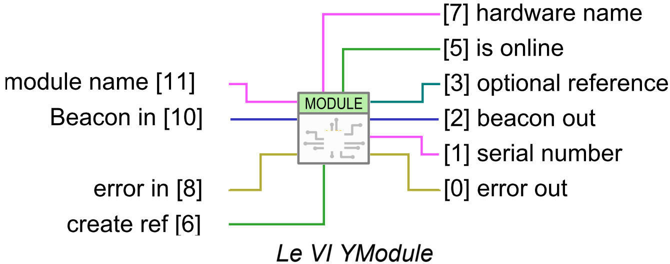
Vous trouverez la liste des fonctions disponibles sur votre Yocto-SDI12 au chapitre *Programmation, concepts généraux*.

Si la fonction recherchée (paramètre *name*) n'est pas accessible, cela ne génèrera pas d'erreur mais la sortie *is online* contiendra FALSE et toutes les sorties contiendront les valeurs "N/A" quand c'est possible. Si la fonction recherchée devient disponible plus tard dans la vie de votre programme, *is online* passera à TRUE.

Si le paramètre *name* contient une chaîne vide, le VI ciblera la première fonction disponible du même type qu'il trouvera. Si aucune fonction n'est disponible, *is online* contiendra FALSE.

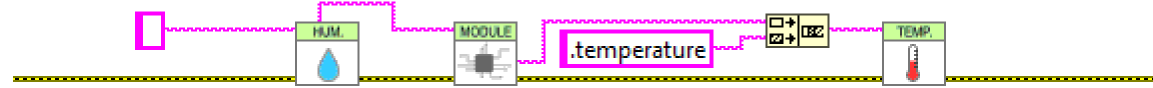
Le VI YModule

Le module `YModule` permet d'interfacer la partie "module" de chaque module Yoctopuce. Il permet de piloter la balise du module et de connaître le numéro de série d'un module.



Le VI YModule

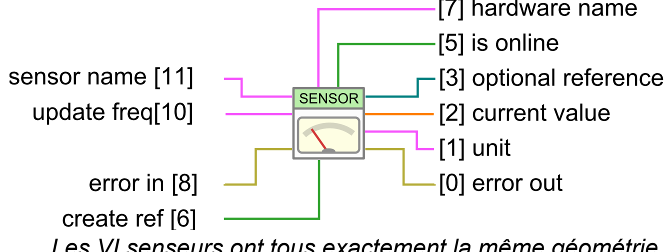
L'entrée *name* fonctionne de manière légèrement différente des autres VIs. S'il est appelé avec le paramètre *name* correspondant à un nom de fonction, le VI `YModule` trouvera la fonction *Module* du module hébergeant la fonction. Il est donc possible de trouver facilement le numéro de série du module d'une fonction quelconque. Cela permet de construire le nom d'autres fonctions qui se trouveraient sur le même module. L'exemple ci dessous trouve la première fonction *YHumidity* disponible et construit le nom de la fonction *YTemperature* qui se trouve sur le même module. Les exemples fournis avec l'API Yoctopuce font un usage extensif de cette technique.



Utilisation du VI YModule pour retrouver les fonctions hébergés sur le même module

Les VI senseurs

Tous les VI correspondant à des senseurs Yoctopuce ont exactement la même géométrie. Les deux sorties permettent de récupérer la valeur mesurée par le capteur correspondant ainsi que l'unité utilisée.



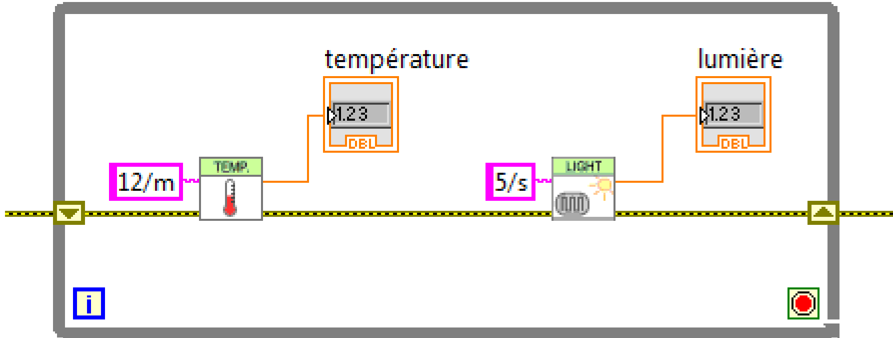
Les VI senseurs ont tous exactement la même géométrie

Le paramètre d'entrée *update freq* est une chaîne de caractères qui permet de configurer la façon dont la valeur de sortie est mis à jour:

- "auto" : la valeur du VI est mise à jour dès que le capteur détecte un changement significatif de valeur. C'est le fonctionnement par défaut.

- "x/s": la valeur du VI est mise à jour *x* fois par seconde avec la valeur instantanée du capteur.
- "x/m","x/h": la valeur du VI est mise à jour *x* fois par minute, (resp. heure) avec la valeur moyenne sur la dernière période. Attention les fréquences maximum sont (60/m) et (3600/h), pour des fréquence plus élevés utiliser la syntaxe (x/s).

La fréquence de mise à jour du VI est un paramètre géré par le module Yoctopuce physique. Si plusieurs VI essayent de changer la fréquence d'un même capteur, la configuration retenue sera celle du dernier appel. Par contre, il est tout à fait possible de configurer des fréquences différentes pour des capteurs du même module Yoctopuce.

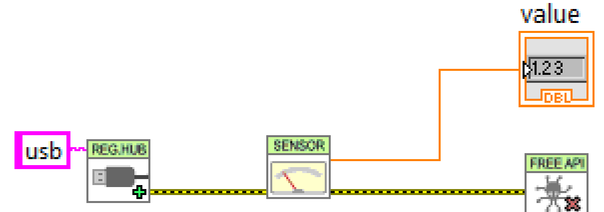


Changement de la fréquence de mise à jour du même module

La fréquence de mise à jour du VI est complètement indépendante de la fréquence d'échantillonnage du capteur qui n'est généralement pas modifiable. Il est inutile et contre-productif de définir une fréquence de mise à jour supérieure à la fréquence d'échantillonnage du capteur.

12.5. Fonctionnement et utilisation des VIs

Voici un exemple parmi les plus simples de VI utilisant l'API Yoctopuce.

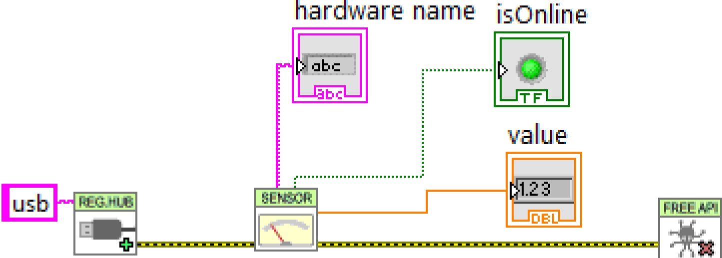


Exemple minimal d'utilisation de l'API Yoctopuce pour LabVIEW

Cet exemple s'appuie sur le VI `YSensor` qui est un VI générique qui permet d'interfacer n'importe quelle fonction senseur d'un module Yoctopuce. Vous pouvez remplacer ce VI par n'importe quel autre de l'API Yoctopuce, ils ont tous la même géométrie et fonctionnent tous de la même manière. Cet exemple se contente de faire trois choses:

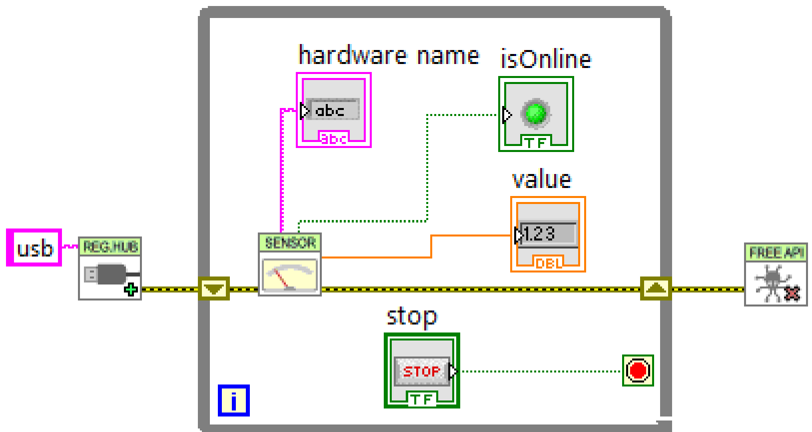
1. Il initialise l'API en mode natif ("usb") avec le VI `YRegisterHub`
2. Il affiche la valeur du premier capteur Yoctopuce qu'il trouve à l'aide du VI `YSensor`
3. Il libère l'API grâce au VI `YFreeAPI`

Cet exemple cherche automatiquement un senseur disponible, si un tel senseur est trouvé on pourra connaitre son nom via la sortie *hardware name* et la sortie *isOnline* sera à TRUE. Si aucun senseur n'est disponible, le VI ne génèrera pas d'erreur mais émulera un senseur fantôme qui sera "offline". Par contre si plus tard, dans la vie de l'application, un senseur devient disponible parce qu'il à été branché, *isOnline* passera à TRUE et le *hardware name* contiendra le nom du capteur. On peut donc facilement ajouter quelques indicateurs à l'exemple précédent pour savoir comment se passe l'exécution.



Utilisation des sorties hardware name et isOnline

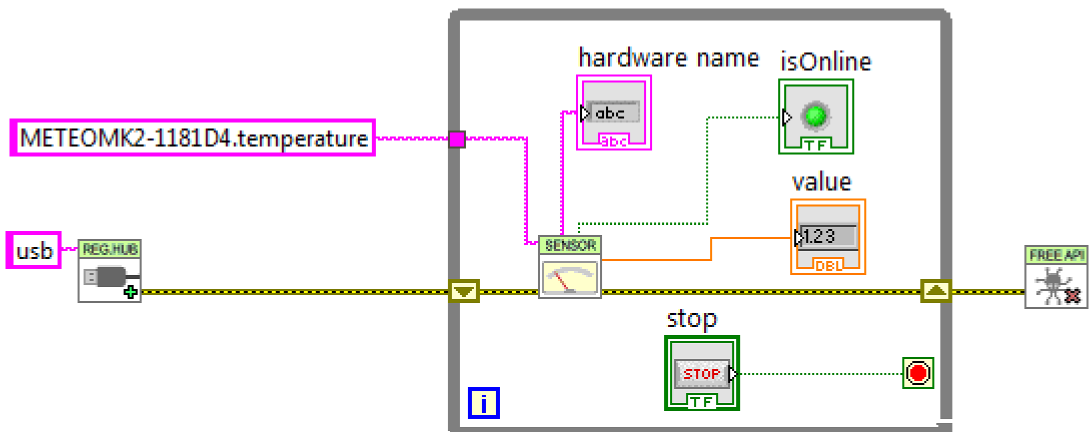
Les VIs de l'API Yoctopuce ne sont qu'une porte d'entrée sur la mécanique interne de la librairie Yoctopuce. Cette mécanique fonctionne indépendamment des VIs Yoctopuce. En effet, la plupart des communications avec les modules électroniques sont gérées automatiquement en arrière plan. C'est pourquoi vous n'avez pas forcément besoin de prendre de précaution particulière pour utiliser les VI Yoctopuce, vous pouvez par exemple les utiliser dans une boucle non temporisée sans que cela pose de problème particulier à l'API.



Les VIs Yoctopuce peuvent être utilisés dans une boucle non temporisée

Notez que le VI `YRegisterHub` n'est pas dans la boucle. Le VI `YRegisterHub` sert à l'initialiser l'API, donc à moins que vous n'ayez plusieurs url à enregistrer, il n'est pas souhaitable de l'appeler plusieurs fois.

Lorsque que le paramètre *name* est initialisé à une chaîne vide, les VI Yoctopuce recherchent automatiquement la fonction avec laquelle ils peuvent travailler, ce qui est très pratique lorsqu'on sait qu'il n'y a qu'une seule fonction du même type disponible que qu'on ne souhaite pas se soucier de gérer som nom. Si le paramètre *name* contient un nom matériel ou un nom logique, le VI cherchera la fonction correspondante, si il ne la trouve pas il émulera une fonction qui sera *offline* en attendant que la vraie fonction devienne disponible.

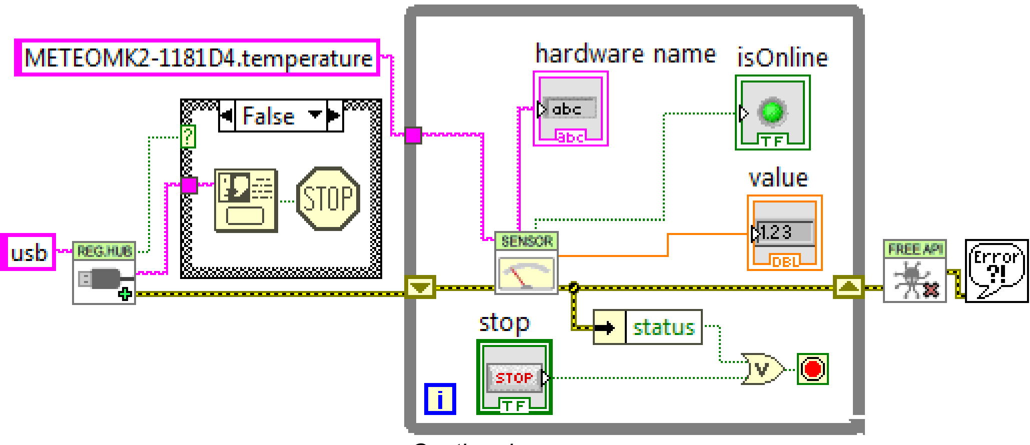


Utilisation de noms pour identifier les fonctions à utiliser

Gestion des erreurs

L'API Yoctopuce pour LabVIEW est codée pour gérer les erreurs d'une manière aussi gracieuse que possible: par exemple si vous utilisez un VI pour accéder à une fonction qui n'existe pas, sa sortie *isOnline* sera à FALSE, les autres sorties seront affecté à *NaN* et les entrées n'auront pas d'effet. Les erreurs fatales sont propagée à travers le canal traditionnel *error in*, *error out*.

Cependant, le VI `YRegisterHub` gère les erreurs de connexion de manière un peu différente. Afin de les rendre plus faciles à gérer, les erreurs de connexions sont signalées à l'aide de sorties *Success* et *error msg*. Si un problème apparait lors de l'appel au VI `YRegisterHub`, *success* contiendra FALSE et *error msg* contiendra une description de l'erreur.



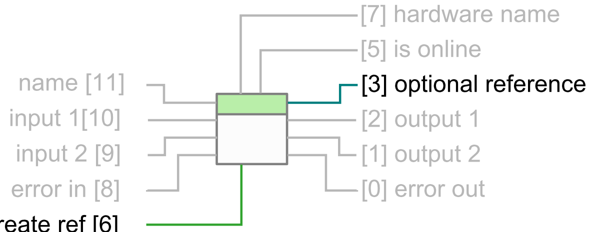
Gestion des erreurs

Le message d'erreur le plus courant est "*Another process is already using yAPI*". Il signifie qu'une autre application, LabVIEW ou autre, utilise déjà l'API en module USB natif. En effet, pour des raison techniques, l'API USB native ne peut être utilisée que par une seule application à la fois sur la même machine. Cette limitation peut être facilement contourné en utilisant le mode réseau.

12.6. Utilisation des objets *Proxy*

L'API Yoctopuce contient des centaines de méthodes, fonctions et propriétés. Il n'était ni possible, ni souhaitable de créer un VI pour chacune d'entre elles. C'est pourquoi il y a un VI par classe qui expose les deux propriétés que Yoctopuce a jugé les plus utiles, mais cela ne veut pas dire que les autres ne sont pas accessibles.

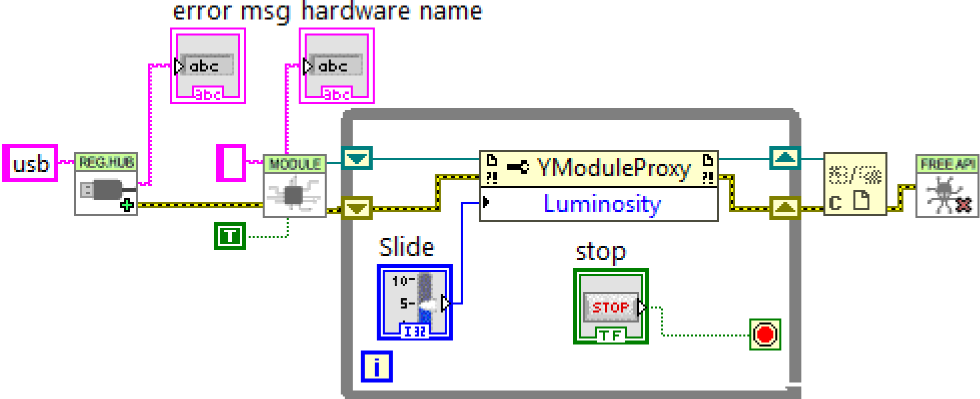
Chaque VI correspondant à une classe dispose de deux connecteurs *create ref* et *optional ref* qui permettent d'obtenir une référence sur l'objet *Proxy* de l'API *.NET Proxy* sur laquelle est construite la librairie LabVIEW.



Les connecteurs pour obtenir une référence sur l'objet Proxy correspondant au VI

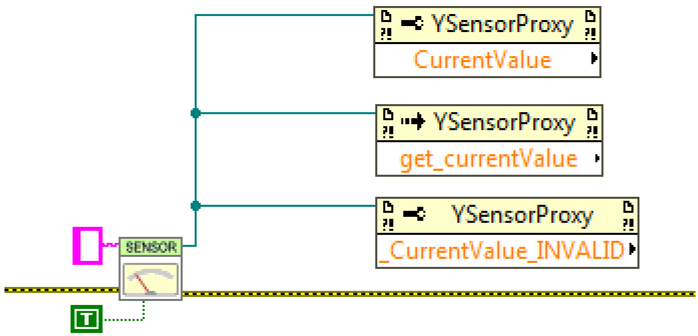
Pour obtenir cette référence, il suffit de mettre *optional ref* à TRUE. Attention, il est impératif de fermer toute référence créée de cette manière, sous peine de saturer rapidement la mémoire de l'ordinateur.

Voici un exemple qui utilise cette technique pour modifier la luminosité des LEDs d'un module Yoctopuce



Contrôle de la luminosité des LEDs d'un module

Notez que chaque référence permet d'obtenir aussi bien des propriétés (noeud *property*) que des méthodes (noeud *invoke*). Par convention, les propriétés sont optimisées pour générer un minimum de communication avec les modules, c'est pourquoi il est recommandé de les utiliser plutôt les méthodes *get_xxx* et *set_xxx* correspondantes qui pourraient sembler équivalentes mais qui ne sont pas optimisées. Les propriétés permettent aussi récupérer les différentes constantes de l'API, qui sont préfixées avec le caractère "_". Pour des raisons techniques, les méthodes *get_xxx* et *set_xxx* ne sont pas toutes disponibles sous forme de propriétés.

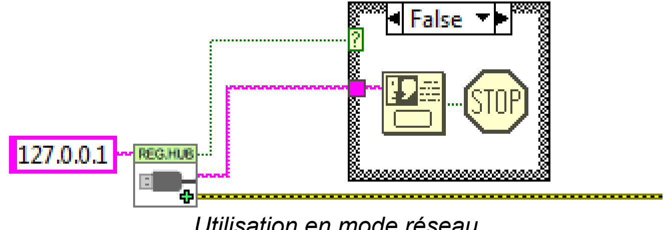


Noeuds Property et Invoke: Utilisation de propriétés, méthodes et constantes

Vous trouverez la description de toutes les propriétés, fonctions et méthodes disponibles dans la documentation de l'API *.NET Proxy*.

Utilisation en réseau

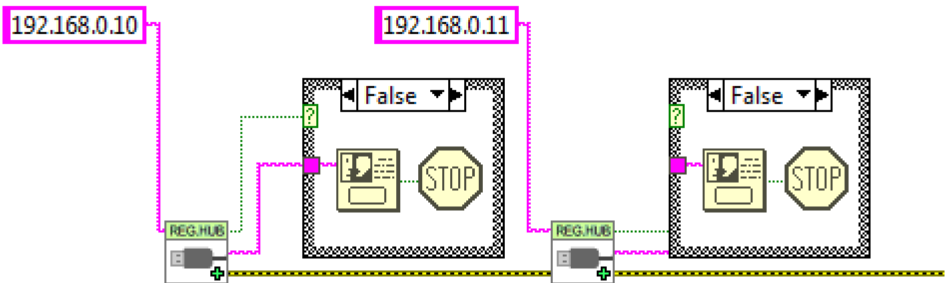
Sur une même machine, il ne peut y avoir qu'un seul processus qui accède aux modules Yoctopuce locaux directement par USB (url mis à "`usb`"). Par contre, plusieurs processus peuvent se connecter en parallèle à des YoctoHubs[7] ou à une machine sur laquelle tourne le logiciel *VirtualHub*[8], y compris la machine locale. Si vous utilisez l'adresse réseau locale de votre machine (127.0.0.1) et qu'un VirtualHub tourne dessus, vous pourrez ainsi contourner la limitation qui empêche l'utilisation en parallèle de l'API native USB.



Utilisation en mode réseau

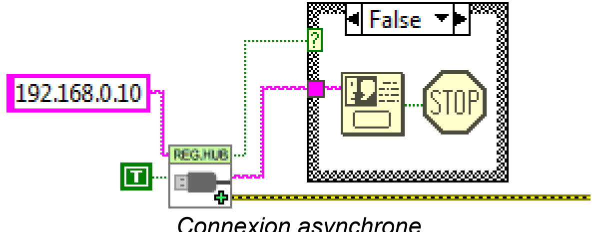
[7] www.yoctopuce.com/FR/products/category/extensions-et-reseau
[8] www.yoctopuce.com/FR/virtualhub.php

Il n'y a pas non plus de limitation sur le nombre d'interfaces réseau auxquels l'API peut se connecter en parallèle. Autrement dit, il est tout à fait possible de faire des appels multiples au VI `YRegisterHub`. C'est le seul cas où il y a un intérêt à appeler le VI `YRegisterHub` plusieurs fois au cours de la vie de l'application.



Les connexions réseau multiples sont possibles

Par défaut, le VI `YRegisterHub` essaye se connecter sur l'adresse donnée en paramètre et génère une erreur (*success*=FALSE) s'il n'y arrive pas parce que personne ne répond. Mais si le paramètre *async* est initialisé à TRUE, aucune erreur ne sera générée en cas d'erreur de connexion, mais si la connexion devient possible plus tard dans la vie de l'application, les modules correspondants seront automatiquement accessibles.



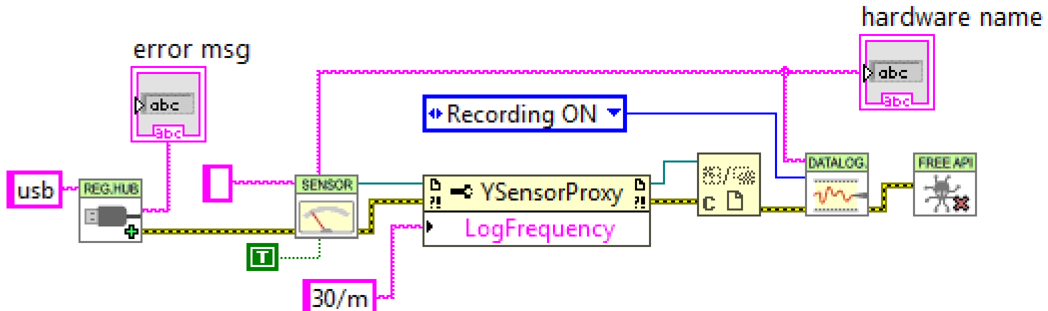
Connexion asynchrone

12.7. Gestion du datalogger

Quasiment tous les senseurs Yoctopuce disposent d'un enregistreur de données qui permet de stocker les mesures des senseurs dans la mémoire non volatile du module. La configuration de l'enregistreur de données peut être réalisée avec le VirtualHub, mais aussi à l'aide d'un peu de code LabVIEW

Enregistrement

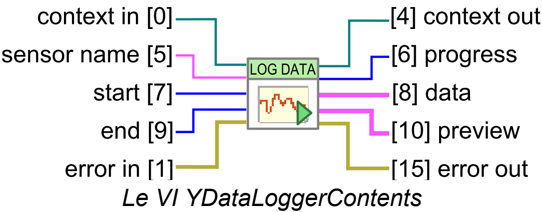
Pour ce faire, il faut configurer la fréquence d'enregistrement en utilisant la propriété "LogFrequency" que l'on atteint avec une référence sur l'objet *Proxy* du senseur utilisé, puis il faut mettre en marche l'enregistreur grâce au VI `YDataLogger`. Noter qu'à la manière du VI `YModule`, le VI `YDataLogger` correspondant à un module peut être obtenu avec son propre nom, mais aussi avec le nom de n'importe laquelle des fonctions présentes sur le même module.



Enclenchement de l'enregistrement de données dans le datalogger

Lecture

La récupération des données de l'enregistreur se fait l'aide du VI `YDataLoggerContents`.

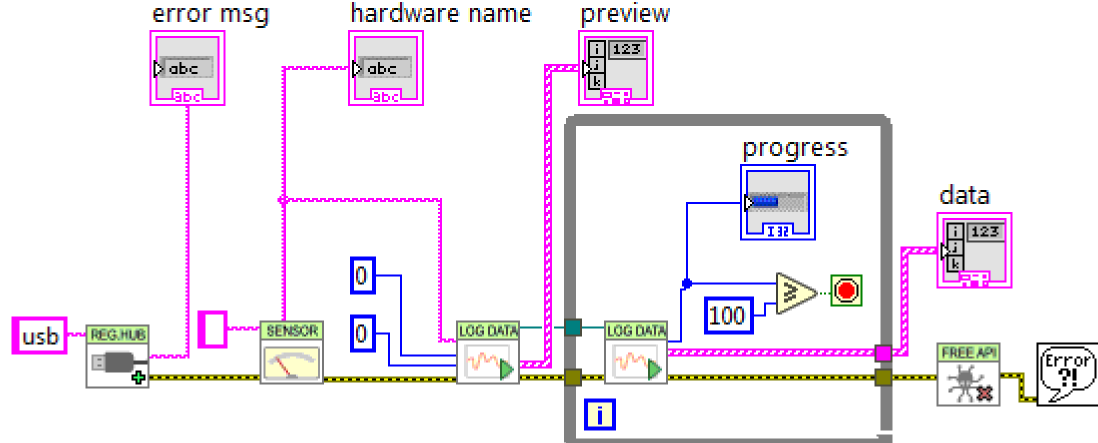


Le VI YDataLoggerContents

Extraire les données de l'enregistreur d'un module Yoctopuce est un processus lent qui peut prendre plusieurs dizaines de secondes. C'est pourquoi le VI qui permet cette opération a été conçu pour fonctionner de manière itérative.

Dans un premier temps le VI doit être appelé avec un nom de senseur, une date de début et une date de fin (timestamp UNIX en UTC). Le couple (0,0) permet d'obtenir la totalité du contenu de l'enregistreur. Ce premier appel permet d'obtenir un résumé du contenu du datalogger et un contexte.

Dans un deuxième temps, il faut rappeler le VI `YDataLoggerContents` en boucle avec le paramètre contexte, jusqu'à ce que la sortie *progress* atteigne la valeur 100. A ce moment la sortie data représente le contenu de l'enregistreur



Récupération du contenu de l'engistreur de données

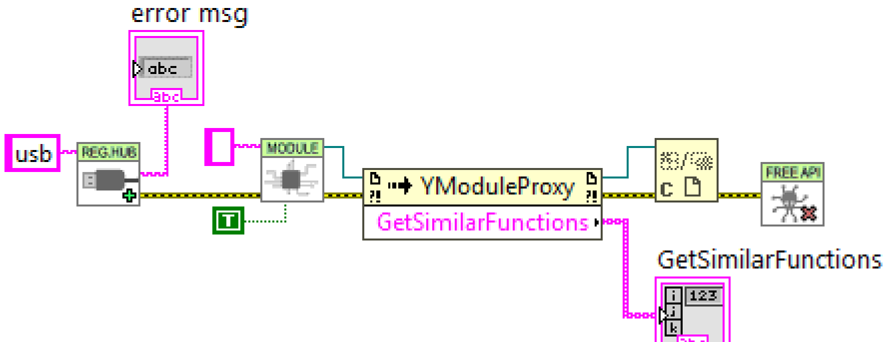
Les résultats et le résumé sont rendus sous la forme d'un tableau de structures qui contiennent les champs suivants:

- *startTime*: début de la période de mesure
- *endTime*: fin de la période de mesure
- *averageValue*: valeur moyenne pour la période
- *minValue*: valeur minimum sur la période
- *maxValue*: valeur maximum sur la période

Notez que si la fréquence d'enregistrement est supérieure à 1 Hz, l'enregistreur ne mémorise que des valeurs instantanées, dans ce cas *averageValue*, *minValue*, et *maxValue* auront la même valeur.

12.8. Énumération de fonctions

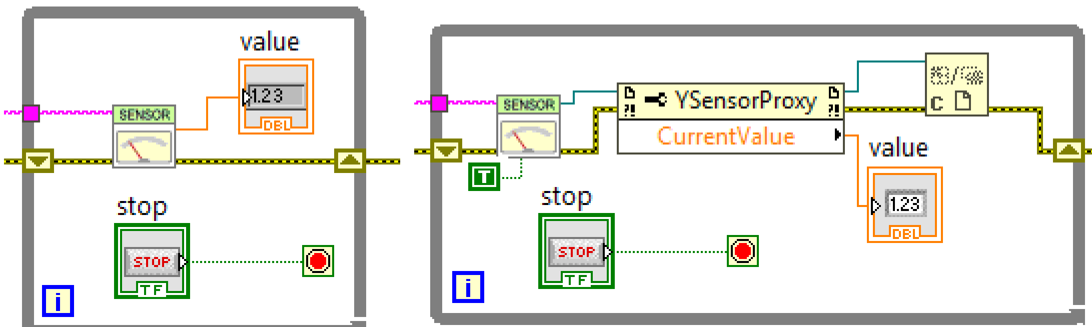
Chaque VI correspondant à un objet de l'API *.NET Proxy* permet de faire une énumération de toutes les fonctions de la même classe via la méthode *getSimilarfunctions()* de l'objet *Proxy* correspondant. Ainsi il est ainsi aisé de faire un inventaire de tous les modules connectés, de tous les capteurs connectés, de tous les relais connectés, etc....



Récupération de la liste de tous les modules connectés

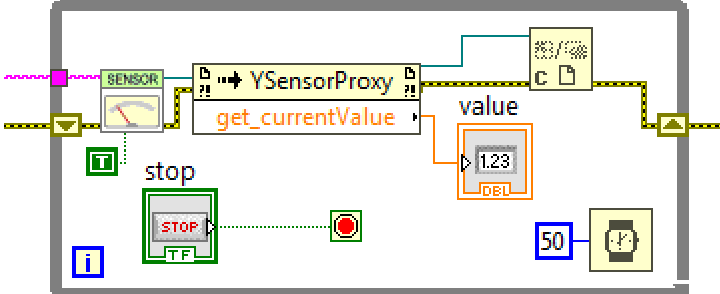
12.9. Un mot sur les performances

L'API Yoctopuce pour LabVIEW été optimisée de manière à ce que les tous les VIs et les propriétés de objets *Proxy* génèrent un minimum de communication avec les modules Yoctopuce. Ainsi vous pouvez les utiliser dans des boucles sans prendre de précaution particulière: vous n'êtes pas *obligés* de ralentir les boucles avec un timer.



Ces deux boucles génèrent peu de communications USB et n'ont pas besoin d'être ralenties

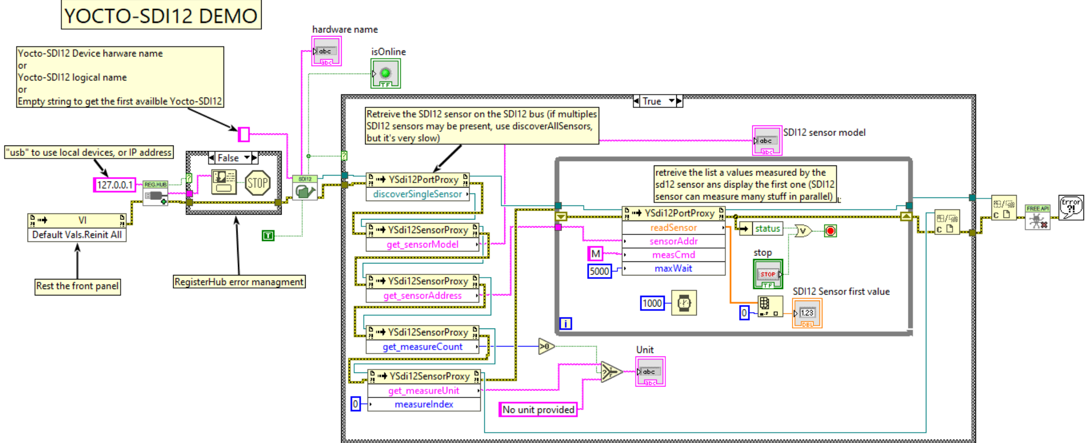
En revanche, presque toutes les méthodes des objets Proxy disponibles vont générer une communication avec les modules Yoctopuce à chaque fois qu'elles seront appelées, il conviendra donc d'éviter de les appeler trop souvent inutilement.



Cette boucle, qui utilise une méthode, doit être ralentie

12.10. Un exemple complet de programme LabVIEW

Voici un exemple qui utilise un Yocto-SDI12 pour interfacer un capteur SHT25 de Sensirion. Après un appel a *RegisterHub*, le bus I2C est alimenté en 3.3V puis le capteur SHT25 est interrogé à l'aide de la fonction *I2CSendAndReceiveBin* pour connaitre la température et le taux d'humidité. Les valeurs lues sont converties en °C et %RH. Les noeuds *I2CSendAndReceiveBin* sont créés grâce à une référence obtenue du VI *YI2cPort*. Notez que cette référence est fermée après utilisation. Une fois l'application terminée, l'API est libérée à l'aide de du VI *YFreeAPI*.



Lire un capteur Sensirion SHT25 avec un Yocto-SDI12

Si vous lisez cette documentation sur un écran, vous pouvez zoomer sur l'image ci-dessus. Vous pourrez aussi retrouver cet exemple dans la librairie Yoctopuce pour LabVIEW

12.11. Différences avec les autres API Yoctopuce

Yoctopuce fait tout son possible pour maintenir une forte cohérence entre les différentes librairies de programmation. Cependant, LabVIEW étant un environnement clairement à part, il en résulte des différences importantes avec les autres librairies.

Ces différences ont aussi été introduites pour rendre l'utilisation des modules aussi facile et intuitive que possible en nécessitant un minimum de code LabVIEW.

YFreeAPI

Contrairement aux autres langages, il est indispensable de libérer l'API native en appelant le VI `YFreeApi` lorsque votre code n'a plus besoin d'utiliser l'API. Si cet appel est omis, l'API native risque de rester bloquée pour les autres applications tant que LabVIEW ne sera pas complètement fermé.

Propriétés

Contrairement aux classes des autres API, les classes disponibles dans LabVIEW implémentent des *propriétés*. Par convention, ces propriétés sont optimisées pour générer un minimum de communication avec les modules tout en se rafraichissant automatiquement. En revanche, les méthodes de type *get_xxx* et *set_xxx* génèrent systématiquement des communications avec les modules Yoctopuce et doivent être appelées à bon escient.

Callback vs Propriétés

Il n'y a pas de callbacks dans l'API Yoctopuce pour LabVIEW, les VIs se rafraichissenti automatiquement: ils sont basés sur les propriétés des objets de l'API *.NET Proxy*.

13. Utilisation du Yocto-SDI12 en Java

Java est un langage orienté objet développé par Sun Microsystem. Son principal avantage est la portabilité, mais cette portabilité a un coût. Java fait une telle abstraction des couches matérielles qu'il est très difficile d'interagir directement avec elles. C'est pourquoi l'API java standard de Yoctopuce ne fonctionne pas en natif: elle doit passer par l'intermédiaire de VirtualHub pour pouvoir communiquer avec les modules Yoctopuce.

13.1. Préparation

Connectez vous sur le site de Yoctopuce et téléchargez les éléments suivants:

- La librairie de programmation pour Java[1]
- VirtualHub[2] pour Windows, macOS ou Linux selon l'OS que vous utilisez

La librairie est disponible en fichier sources, mais elle aussi disponible sous la forme d'un fichier jar. Branchez vos modules, Décompressez les fichiers de la librairie dans un répertoire de votre choix. Lancez VirtualHub et vous pouvez commencer vos premiers tests. Vous n'avez pas besoin d'installer de driver.

Afin de les garder simples, tous les exemples fournis dans cette documentation sont des applications consoles. Il va de soit que que le fonctionnement des librairies est strictement identiques si vous les intégrez dans une application dotée d'une interface graphique.

13.2. Contrôle de la fonction Sdi12Port

Il suffit de quelques lignes de code pour piloter un Yocto-SDI12. Voici le squelette d'un fragment de code Java qui utilise la fonction Sdi12Port.

```
[...]
// On active l'accès aux modules locaux à travers le VirtualHub
YAPI.RegisterHub("127.0.0.1");
[...]

// On récupère l'objet permettant d'intéragir avec le module
sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.sdi12Port");

// Pour gérer le hot-plug, on vérifie que le module est là
if (sdi12port.isOnline())
```

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

```
{
    // Utiliser sdi12port.set_sdi12Mode()
    [...]
}

[...]
```

Voyons maintenant en détail ce que font ces quelques lignes.

YAPI.RegisterHub

La fonction `YAPI.RegisterHub` initialise l'API de Yoctopuce en indiquant où les modules doivent être recherchés. Le paramètre est l'adresse du virtual hub capable de voir les modules. Si l'initialisation se passe mal, une exception sera générée.

YSdi12Port.FindSdi12Port

La fonction `YSdi12Port.FindSdi12Port` permet de retrouver un port SDI12 en fonction du numéro de série de son module hôte et de son nom de fonction. Mais vous pouvez tout aussi bien utiliser des noms logiques que vous auriez préalablement configurés. Imaginons un module Yocto-SDI12 avec le numéros de série *YSDIMK01-123456* que vous auriez appelé "*MonModule*" et dont vous auriez nommé la fonction *sdi12Port* "*MaFonction*", les cinq appels suivants seront strictement équivalents (pour autant que *MaFonction* ne soit définie qu'une fois, pour éviter toute ambiguïté):

```
sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.sdi12Port")
sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.MaFonction")
sdi12port = YSdi12Port.FindSdi12Port("MonModule.sdi12Port")
sdi12port = YSdi12Port.FindSdi12Port("MonModule.MaFonction")
sdi12port = YSdi12Port.FindSdi12Port("MaFonction")
```

`YSdi12Port.FindSdi12Port` renvoie un objet que vous pouvez ensuite utiliser à loisir pour contrôler le port SDI12.

isOnline

La méthode `isOnline()` de l'objet renvoyé par `YSdi12Port.FindSdi12Port` permet de savoir si le module correspondant est présent et en état de marche.

reset

La méthode `reset()` de l'objet returné par `YSdi12Port.FindSerialPort` vide tous les tampons du port série.

discoverSingleSensor

La méthode `discoverSingleSensor()` cherche l'adresse du capteur connecté sur le port SDI-12 et renvoie un objet avec toute les informations du capteur.

readSensor

La méthode `readSensor()` transmet la commande spécifiée sur le port SDI-12 au capteur spécifié et renvoie une liste d'objet avec toute les valeurs envoyées par le capteur.

Un exemple réel

Lancez votre environnement java et ouvrez le projet correspondant, fourni dans le répertoire **Examples/Doc-GettingStarted-Yocto-SDI12** de la librairie Yoctopuce.

Vous reconnaîtrez dans cet exemple l'utilisation des fonctions expliquées ci-dessus, cette fois utilisées avec le décorum nécessaire à en faire un petit programme d'exemple concret.

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

import java.util.ArrayList;

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);
        }

        YSdi12Port sdi12Port;
        sdi12Port = YSdi12Port.FirstSdi12Port();
        if (sdi12Port == null || !sdi12Port.isOnline()) {
            System.out.println("No module connected (check USB cable)");
            System.exit(1);
        }
        try {
            sdi12Port.reset();
            YSdi12SensorInfo singleSensor = sdi12Port.discoverSingleSensor();
            System.out.println(String.format("%-35s %s " ,"Sensor address :",
singleSensor.get_sensorAddress()));
            System.out.println(String.format("%-35s %s " ,"Sensor SDI-12 compatibility :
" , singleSensor.get_sensorProtocol()));
            System.out.println(String.format("%-35s %s " ,"Sensor company name : " ,
singleSensor.get_sensorVendor()));
            System.out.println(String.format("%-35s %s " ,"Sensor model number : " ,
singleSensor.get_sensorModel()));
            System.out.println(String.format("%-35s %s " ,"Sensor version : " ,
singleSensor.get_sensorVersion()));
            System.out.println(String.format("%-35s %s " ,"Sensor serial number : " ,
singleSensor.get_sensorSerial()));

            ArrayList<Double> valSensor = sdi12Port.readSensor
(singleSensor.get_sensorAddress(),"M",5000);

            for (int i = 0; i < valSensor.size(); i = i+1)
            {
                if (singleSensor.get_measureCount() > 1)
                {
                    System.out.println(String.format("%s : %-6.2f %-10s (%s)",
                            singleSensor.get_measureSymbol(i), valSensor.get(i),
singleSensor.get_measureUnit(i), singleSensor.get_measureDescription(i)));
                }
                else {
                    System.out.print(valSensor.get(i));
                }

            }

        } catch (YAPI_Exception ex) {
            System.out.println("Module not connected (check identification and USB cable)"
);
        }

        YAPI.FreeAPI();
    }
}
```

13.3. Contrôle de la partie module

Chaque module peut-être contrôlé d'une manière similaire, vous trouverez ci-dessous un simple programme d'exemple affichant les principaux paramètres d'un module et permettant d'activer la balise de localisation.

```
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();
    }
}
```

Chaque propriété `xxx` du module peut être lue grâce à une méthode du type `YModule.get_xxxx()`, et les propriétés qui se sont pas en lecture seule peuvent être modifiées à l'aide de la méthode `YModule.set_xxx()` Pour plus de détails concernant ces fonctions utilisées, reportez-vous aux chapitre API

Modifications des réglages du module

Lorsque que vous souhaitez modifier les réglages d'un module, il suffit d'appeler la fonction `YModule.set_xxx()` correspondante, cependant cette modification n'a lieu que dans la mémoire vive du module: si le module redémarre, les modifications seront perdues. Pour qu'elle soient mémorisées de manière persistante, il est nécessaire de demander au module de sauvegarder sa configuration courante dans sa mémoire non volatile. Pour cela il faut utiliser la méthode `YModule.saveToFlash()`. Inversement il est possible de forcer le module à oublier ses réglages courants en utilisant la méthode `YModule.revertFromFlash()`. Ce petit exemple ci-dessous vous permet changer le nom logique d'un 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();
    }
}
```

Attention, le nombre de cycles d'écriture de la mémoire non volatile du module est limité. Passé cette limite plus rien ne garantit que la sauvegarde des réglages se passera correctement. Cette limite, liée à la technologie employée par le micro-processeur du module se situe aux alentour de 100000 cycles. Pour résumer vous ne pouvez employer la fonction `YModule.saveToFlash()` que 100000 fois au cours de la vie du module. Veillez donc à ne pas appeler cette fonction depuis l'intérieur d'une boucle.

Enumeration des modules

Obtenir la liste des modules connectés se fait à l'aide de la fonction `YModule.yFirstModule()` qui renvoie le premier module trouvé, il suffit ensuite d'appeler la mehode `nextModule()` de cet objet pour trouver les modules suivants, et ce tant que la réponse n'est pas un `null`. Ci-dessous un petit exemple listant les module connectés

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

public class Demo {

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

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

13.4. Gestion des erreurs

Lorsque vous implémentez un programme qui doit interagir avec des modules USB, vous ne pouvez pas faire abstraction de la gestion des erreurs. Il y aura forcément une occasion où un utilisateur aura débranché le périphérique, soit avant de lancer le programme, soit même en pleine opération. La librairie Yoctopuce est prévue pour vous aider à supporter ce genre de comportements, mais votre code doit néanmoins être fait pour se comporter au mieux pour interpréter les erreurs signalées par la librairie.

La manière la plus simple de contourner le problème est celle que nous avons employé pour les petits exemples précédents de ce chapitre: avant d'accéder à un module, on vérifie qu'il est en ligne avec la méthode `isOnline()` et on suppose ensuite qu'il va y rester pendant la fraction de seconde nécessaire à exécuter les lignes de code suivantes. Ce n'est pas parfait, mais ça peut suffire dans certains cas. Il faut toutefois être conscient qu'on ne peut pas totalement exclure une erreur se produisant après le `isOnline()`, qui pourrait faire planter le programme.

Dans l'API java, le traitement d'erreur est implémenté au moyen d'exceptions. Vous devrez donc intercepter et traiter correctement ces exceptions si vous souhaitez avoir un projet fiable qui ne crashera pas des que vous débrancherez un module.

14. Utilisation du Yocto-SDI12 avec Android

A vrai dire, Android n'est pas un langage de programmation, c'est un système d'exploitation développé par Google pour les appareils portables tels que smart phones et tablettes. Mais il se trouve que sous Android tout est programmé avec le même langage de programmation: Java. En revanche les paradigmes de programmation et les possibilités d'accès au hardware sont légèrement différentes par rapport au Java classique, ce qui justifie un chapitre à part sur la programmation Android.

14.1. Accès Natif et VirtualHub

Contrairement à l'API Java classique, l'API Java pour Android accède aux modules USB de manière native. En revanche, comme il n'existe pas de VirtualHub tournant sous Android, il n'est pas possible de prendre le contrôle à distance de modules Yoctopuce pilotés par une machine sous Android. Bien sûr, l'API Java pour Android reste parfaitement capable de se connecter à VirtualHub tournant sur un autre OS.

14.2. Préparation

Connectez-vous sur le site de Yoctopuce et téléchargez la librairie de programmation pour Java pour Android[1]. La librairie est disponible en fichiers sources, mais elle aussi disponible sous la forme d'un fichier jar. Branchez vos modules, décompressez les fichiers de la librairie dans le répertoire de votre choix. Et configurez votre environnement de programmation Android pour qu'il puisse les trouver.

Afin de les garder simples, tous les exemples fournis dans cette documentation sont des fragments d'application Android. Vous devrez les intégrer dans vos propres applications Android pour les faire fonctionner. En revanche vous pourrez trouver des applications complètes dans les exemples fournis avec la librairie Java pour Android.

14.3. Compatibilité

Dans un monde idéal, il suffirait d'avoir un téléphone sous Android pour pouvoir faire fonctionner des modules Yoctopuce. Malheureusement, la réalité est légèrement différente, un appareil tournant sous Android doit répondre à un certain nombre d'exigences pour pouvoir faire fonctionner des modules USB Yoctopuce en natif.

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

Version d'Android

Notre librairie peut être compilée pour fonctionner avec les anciennes versions aussi longtemps que les outils Android nous permettent de les supporter, soit environ les versions des dix dernières années.

Support USB *host*

Il faut bien sûr que votre machine dispose non seulement d'un port USB, mais il faut aussi que ce port soit capable de tourner en mode *host*. En mode *host*, la machine prend littéralement le contrôle des périphériques qui lui sont raccordés. Les ports USB d'un ordinateur bureau, par exemple, fonctionnent mode *host*. Le pendant du mode *host* est le mode *device*. Les clefs USB par exemple fonctionnent en mode *device*: elles ne peuvent qu'être contrôlées par un *host*. Certains ports USB sont capables de fonctionner dans les deux modes, ils s'agit de ports *OTG* (*On The Go*). Il se trouve que beaucoup d'appareils portables ne fonctionnent qu'en mode "device": ils sont conçus pour être branchés à chargeur ou un ordinateur de bureau, rien de plus. Il est donc fortement recommandé de lire attentivement les spécifications techniques d'un produit fonctionnant sous Android avant d'espérer le voir fonctionner avec des modules Yoctopuce.

Disposer d'une version correcte d'Android et de ports USB fonctionnant en mode *host* ne suffit malheureusement pas pour garantir un bon fonctionnement avec des modules Yoctopuce sous Android. En effet certains constructeurs configurent leur image Android afin que les périphériques autres que clavier et mass storage soit ignorés, et cette configuration est difficilement détectable. En l'état actuel des choses, le meilleur moyen de savoir avec certitude si un matériel Android spécifique fonctionne avec les modules Yoctopuce consiste à essayer.

14.4. Activer le port USB sous Android

Par défaut Android `nautorise` pas une application à accéder aux périphériques connectés au port USB. Pour que votre application puisse interagir avec un module Yoctopuce branché directement sur votre tablette sur un port USB quelques étapes supplémentaires sont nécessaires. Si vous comptez uniquement interagir avec des modules connectés sur une autre machine par IP, vous pouvez ignorer cette section.

Il faut déclarer dans son `AndroidManifest.xml` l'utilisation de la fonctionnalité "USB Host" en ajoutant le tag `<uses-feature android:name="android.hardware.usb.host" />` dans la section `manifest`.

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

Lors du premier accès à un module Yoctopuce, Android va ouvrir une fenêtre pour informer l'utilisateur que l'application va accéder module connecté. L'utilisateur peut refuser ou autoriser `laccès` au périphérique. Si l'utilisateur accepte, l'application pourra accéder au périphérique connecté jusqu'à la prochaine déconnexion du périphérique. Pour que la librairie Yoctopuce puisse gérer correctement ces autorisations, il faut lui fournir un pointeur sur le contexte de l'application en appelant la méthode EnableUSBHost de la classe YAPI avant le premier accès USB. Cette fonction prend en argument un objet de la classe `android.content.Context` (ou d'une sous-classe). Comme la classe Activity est une sous-classe de `Context`, le plus simple est de d'appeler `YAPI.EnableUSBHost(this);` dans la méthode `onCreate` de votre application. Si l'objet passé en paramètre n'est pas du bon type, une exception `YAPI_Exception` sera générée.

```
...
@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());
        }
}
...
```

Lancement automatique

Il est possible d'enregistrer son application comme application par défaut pour un module USB, dans ce cas des qu'un module sera connecté au système, l'application sera lancée automatiquement. Il faut ajouter *<action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"/>* dans la section *<intent-filter>* de l'activité principale. La section *<activity>* doit contenir un pointeur sur un fichier xml qui contient la liste des modules USB qui peuvent lancer l'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>
```

Le fichier XML qui contient la liste des modules qui peuvent lancer l'application doit être sauvé dans le répertoire res/xml. Ce fichier contient une liste de *vendorId* et *deviceID* USB en décimal. L'exemple suivant lance l'application dès qu'un Yocto-Relay ou un Yocto-PowerRelay est connecté. Vous pouvez trouver le vendorId et deviceId des modules Yoctopuce dans la section caractéristiques de la documentation.

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

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

14.5. Contrôle de la fonction Sdi12Port

Il suffit de quelques lignes de code pour piloter un Yocto-SDI12. Voici le squelette d'un fragment de code Java qui utilise la fonction Sdi12Port.

```
[...]
// On active la détection des modules sur USB
YAPI.EnableUSBHost(this);
YAPI.RegisterHub("usb");
[...]
// On récupère l'objet permettant de communiquer avec le module
sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.sdi12Port");

// Pour gérer le hot-plug, on vérifie que le module est là
if (sdi12port.isOnline())
{
    // Utilisez sdi12port.set_sdi12Mode()
    [...]
}
```

```
[...]
```

Voyons maintenant en détail ce que font ces quelques lignes.

YAPI.EnableUSBHost

La fonction `YAPI.EnableUSBHost` initialise l'API avec le Context de l'application courante. Cette fonction prend en argument un objet de la classe `android.content.Context` (ou d'une sous-classe). Si vous comptez uniquement vous connecter à d'autres machines par IP vous cette fonction est factultative.

YAPI.RegisterHub

La fonction `YAPI.RegisterHub` initialise l'API de Yoctopuce en indiquant où les modules doivent être recherchés. Le paramètre est l'adresse du virtual hub capable de voir les modules. Si l'on passe la chaine de caractère `"usb"`, l'API va travailler avec les modules connectés localement à la machine. Si l'initialisation se passe mal, une exception sera générée.

YSdi12Port.FindSdi12Port

La fonction `YSdi12Port.FindSdi12Port` permet de retrouver un port SDI12 en fonction du numéro de série de son module hôte et de son nom de fonction. Mais vous pouvez tout aussi bien utiliser des noms logiques que vous auriez préalablement configurés. Imaginons un module Yocto-SDI12 avec le numéros de série *YSDIMK01-123456* que vous auriez appelé "*MonModule*" et dont vous auriez nommé la fonction *sdi12Port* "*MaFonction*", les cinq appels suivants seront strictement équivalents (pour autant que *MaFonction* ne soit définie qu'une fois, pour éviter toute ambiguïté):

```
sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.sdi12Port")
sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.MaFonction")
sdi12port = YSdi12Port.FindSdi12Port("MonModule.sdi12Port")
sdi12port = YSdi12Port.FindSdi12Port("MonModule.MaFonction")
sdi12port = YSdi12Port.FindSdi12Port("MaFonction")
```

`YSdi12Port.FindSdi12Port` renvoie un objet que vous pouvez ensuite utiliser à loisir pour contrôler le port SDI12.

isOnline

La méthode `isOnline()` de l'objet renvoyé par `YSdi12Port.FindSdi12Port` permet de savoir si le module correspondant est présent et en état de marche.

reset

La méthode `reset()` de l'objet returné par `YSdi12Port.FindSerialPort` vide tous les tampons du port série.

discoverSingleSensor

La méthode `discoverSingleSensor()` cherche l'adresse du capteur connecté sur le port SDI-12 et renvoie un objet avec toute les informations du capteur.

readSensor

La méthode `readSensor()` transmet la commande spécifiée sur le port SDI-12 au capteur spécifié et renvoie une liste d'objet avec toute les valeurs envoyées par le capteur.

Un exemple réel

Lancez votre environnement java et ouvrez le projet correspondant, fourni dans le répertoire **Examples/Doc-Examples** de la librairie Yoctopuce.

Vous reconnaîtrez dans cet exemple l'utilisation des fonctions expliquées ci-dessus, cette fois utilisées avec le décorum nécessaire à en faire un petit programme d'exemple concret.

```
package com.yoctopuce.doc_examples;

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

import com.yoctopuce.YoctoAPI.YAPI;
import com.yoctopuce.YoctoAPI.YAPI_Exception;
import com.yoctopuce.YoctoAPI.YSdi12Port;
import com.yoctopuce.YoctoAPI.YSdi12SensorInfo;

import java.util.ArrayList;

public class GettingStarted_Yocto_SDI12 extends Activity implements OnItemSelectedListener
{

    private YSdi12Port sdi12Port = null;
    private ArrayAdapter<String> aa;
    private YSdi12SensorInfo singleSensor = null;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.gettingstarted_yocto_sdi12);
        Spinner my_spin = (Spinner) findViewById(R.id.spinner1);
        my_spin.setOnItemSelectedListener(this);
        aa = new ArrayAdapter<String>(this, android.R.layout.simple_spinner_item);
        aa.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
        my_spin.setAdapter(aa);
    }

    @Override
    protected void onStart() {
        super.onStart();
        aa.clear();
        try {
            YAPI.EnableUSBHost(this);
            YAPI.RegisterHub("usb");
            YSdi12Port s = YSdi12Port.FirstSdi12Port();
            while (s != null) {
                String hwid = s.get_hardwareId();
                aa.add(hwid);
                s = s.nextSdi12Port();
            }
        } catch (YAPI_Exception e) {
            e.printStackTrace();
        }
        aa.notifyDataSetChanged();
    }

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

    @Override
    public void onItemSelected(AdapterView<?> parent, View view, int pos, long id) {
        String hwid = (String) parent.getItemAtPosition(pos);
        sdi12Port = YSdi12Port.FindSdi12Port(hwid);
        try {
            sdi12Port.reset();
            singleSensor = sdi12Port.discoverSingleSensor();

        } catch (YAPI_Exception e) {
            TextView textView = (TextView) findViewById(R.id.response);
            textView.setText(e.getStackTraceToString());
        }
    }

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

```
    /**
     * Called when the user touches the button State A
     */
    public void update(View view) {
        TextView textView = (TextView) findViewById(R.id.response);
        StringBuilder response = new StringBuilder();
        if (sdi12Port == null || singleSensor == null) {
            textView.setText("No module connected (check USB cable)");
            return;
        }
        try {
            response.append(String.format("%-35s %s ", "Sensor address :",
singleSensor.get_sensorAddress())).append("\n");
            response.append(String.format("%-35s %s ", "Sensor SDI-12 compatibility : ",
singleSensor.get_sensorProtocol())).append("\n");
            response.append(String.format("%-35s %s ", "Sensor company name : ",
singleSensor.get_sensorVendor())).append("\n");
            response.append(String.format("%-35s %s ", "Sensor model number : ",
singleSensor.get_sensorModel())).append("\n");
            response.append(String.format("%-35s %s ", "Sensor version : ",
singleSensor.get_sensorVersion())).append("\n");
            response.append(String.format("%-35s %s ", "Sensor serial number : ",
singleSensor.get_sensorSerial())).append("\n");
            ArrayList<Double> valSensor = sdi12Port.readSensor
(singleSensor.get_sensorAddress(), "M", 5000);
            for (int i = 0; i < valSensor.size(); i = i + 1) {
                if (singleSensor.get_measureCount() > 1) {
                    response.append(String.format("%s : %-6.2f %-10s (%s)\n",
                            singleSensor.get_measureSymbol(i), valSensor.get(i),
                            singleSensor.get_measureUnit(i),
singleSensor.get_measureDescription(i)));
                } else {
                    response.append(String.format("%f\n", valSensor.get(i)));
                }
            }
        } catch (YAPI_Exception e) {
            response.append(e.getStackTraceToString());
        }
        textView.setText(response.toString());
    }
}
```

14.6. Contrôle de la partie module

Chaque module peut-être contrôlé d'une manière similaire, vous trouverez ci-dessous un simple programme d'exemple affichant les principaux paramètres d'un module et permettant d'activer la balise de localisation.

```
package com.yoctopuce.doc_examples;

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

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

public class ModuleControl extends Activity implements OnItemSelectedListener
{

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

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

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

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

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

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

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

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

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

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

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

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

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

Chaque propriété `xxx` du module peut être lue grâce à une méthode du type `YModule.get_xxxx()`, et les propriétés qui se sont pas en lecture seule peuvent être modifiées à l'aide de la méthode `YModule.set_xxx()` Pour plus de détails concernant ces fonctions utilisées, reportez-vous aux chapitre API

Modifications des réglages du module

Lorsque que vous souhaitez modifier les réglages d'un module, il suffit d'appeler la fonction `YModule.set_xxx()` correspondante, cependant cette modification n'a lieu que dans la mémoire vive du module: si le module redémarre, les modifications seront perdues. Pour qu'elle soient mémorisées de manière persistante, il est nécessaire de demander au module de sauvegarder sa configuration courante dans sa mémoire non volatile. Pour cela il faut utiliser la méthode `YModule.saveToFlash()`. Inversement il est possible de forcer le module à oublier ses réglages courants en utilisant la méthode `YModule.revertFromFlash()`. Ce petit exemple ci-dessous vous permet changer le nom logique d'un module.

```
package com.yoctopuce.doc_examples;

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

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

public class SaveSettings extends Activity implements OnItemSelectedListener
{

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

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

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

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

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

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

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

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

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

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

```
}
```

Attention, le nombre de cycles d'écriture de la mémoire non volatile du module est limité. Passé cette limite plus rien ne garantit que la sauvegarde des réglages se passera correctement. Cette limite, liée à la technologie employée par le micro-processeur du module se situe aux alentour de 100000 cycles. Pour résumer vous ne pouvez employer la fonction `YModule.saveToFlash()` que 100000 fois au cours de la vie du module. Veillez donc à ne pas appeler cette fonction depuis l'intérieur d'une boucle.

Enumeration des modules

Obtenir la liste des modules connectés se fait à l'aide de la fonction `YModule.yFirstModule()` qui renvoie le premier module trouvé, il suffit ensuite d'appeler la mehode `nextModule()` de cet objet pour trouver les modules suivants, et ce tant que la réponse n'est pas un `null`. Ci-dessous un petit exemple listant les module connectés

```
package com.yoctopuce.doc_examples;

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

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

public class Inventory extends Activity
{

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

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

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

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

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

}
```

14.7. Gestion des erreurs

Lorsque vous implémentez un programme qui doit interagir avec des modules USB, vous ne pouvez pas faire abstraction de la gestion des erreurs. Il y aura forcément une occasion où un utilisateur aura débranché le périphérique, soit avant de lancer le programme, soit même en pleine opération. La librairie Yoctopuce est prévue pour vous aider à supporter ce genre de comportements, mais votre code doit néanmoins être fait pour se comporter au mieux pour interpréter les erreurs signalées par la librairie.

La manière la plus simple de contourner le problème est celle que nous avons employé pour les petits exemples précédents de ce chapitre: avant d'accéder à un module, on vérifie qu'il est en ligne avec la méthode `isOnline()` et on suppose ensuite qu'il va y rester pendant la fraction de seconde nécessaire à exécuter les lignes de code suivantes. Ce n'est pas parfait, mais ça peut suffire dans certains cas. Il faut toutefois être conscient qu'on ne peut pas totalement exclure une erreur se produisant après le `isOnline()`, qui pourrait faire planter le programme.

Dans l'API java pour Android, le traitement d'erreur est implémenté au moyen d'exceptions. Vous devrez donc intercepter et traiter correctement ces exceptions si vous souhaitez avoir un projet fiable qui ne crashera pas des que vous débrancherez un module.

15. Utilisation du Yocto-SDI12 en TypeScript

TypeScript est une version améliorée du langage de programmation JavaScript. Il s'agit d'un sur-ensemble syntaxique avec typage fort, permettant d'améliorer la fiabilité du code, mais qui est transcompilé en JavaScript avant l'exécution, pour être ensuite interprêté par n'importe quel navigateur Web ou par Node.js.

Cette librairie de programmation Yoctopuce permet donc de coder des applications JavaScript tout en bénéficiant d'un typage fort. Comme notre librairie EcmaScript, elle utilise les fonctionnalités asynchrones introduites dans la version ECMAScript 2017 et qui sont maintenant disponibles nativement dans tous les environnements JavaScript modernes. Néanmoins, à ce jour, le code TypeScript n'est pas utilisable directement dans un navigateur Web ou Node.js, donc il est nécessaire de le compiler en JavaScript avant l'exécution.

La librairie peut travailler aussi bien dans un navigateur internet que dans un environnement Node.js. Pour satisfaire aux exigences de résolution statique des dépendances, et pour éviter les ambiguïtés qui surgiraient lors de l'utilisation d'environnements hybrides tels qu'Electron, la sélection de l'environnement doit être faite explicitement à l'import de la librairie, en important dans le projet soit `yocto_api_nodejs.js`, soit `yocto_api_html.js`.

La librairie peut être intégrée à vos projets de plusieurs manières, selon ce qui convient le mieux à votre projet:

- en copiant directement les fichiers sources TypeScript de notre librairie dans votre projet, et en les ajoutant à votre script de build. Il suffit en général de peu de fichiers pour couvrir la plupart des utilisations. Vous les trouverez dans le sous-répertoire `src` de notre librairie.
- en utilisant la résolution de modules CommonJS, supportée par TypeScript, avec un gestionnaire de packages comme `npm`. Vous trouverez une version transpilée au standard CommonJS dans le sous-répertoire `dist/cjs` de la librairie, y compris les fichiers de définition de type (extension `.d.ts`) et les fichiers de debug (extension `.js.map`) permettant le traçage des erreurs dans les fichiers sources TypeScript. Nous avons aussi publié ces fichiers sur `npmjs` sous le nom `yoctolib-cjs`.
- en utilisant la résolution de modules ECMAScript 2015, aussi supportée par TypeScript, et utilisable directement depuis une page HTML par un référencement relatif. Vous trouverez une version transpilée en module ECMAScript 2015 dans le sous-répertoire `dist/esm` de la librairie, y compris les fichiers de définition de type (extension `.d.ts`) et les fichiers de debug (extension `.js.map`) permettant le traçage des erreurs dans les fichiers sources TypeScript. Nous avons aussi publié ces fichiers sur `npmjs` sous le nom `yoctolib-esm`.

15.1. Utiliser la librairie Yoctopuce pour TypeScript

1. Commencez par installer TypeScript sur votre machine si cela n'est pas déjà fait. Pour cela:

- Installez sur votre machine de développement une version raisonnablement récente de Node.js (version 10 ou plus récente). Vous pouvez l'obtenir gratuitement sur le site officiel: http://nodejs.org. Assurez vous de l'installer entièrement, y compris `npm`, et de l'ajouter à votre system path.
- Installez ensuite TypeScript sur votre machine à l'aide de la commande:

```
npm install -g typescript
```

2. Connectez-vous ensuite sur le site Web de Yoctopuce et téléchargez les éléments suivants:

- La librairie de programmation pour TypeScript[1]
- Le programme VirtualHub[2] pour Windows, macOS ou Linux selon l'OS que vous utilisez. En effet, TypeScript et JavaScript font partie de ces langages qui ne vous permettront pas d'accéder directement aux périphériques USB. C'est pourquoi si vous désirez travailler avec des modules branchés par USB, vous devrez faire tourner la passerelle de Yoctopuce appelée VirtualHub sur la machine à laquelle sont branchés les modules. Vous n'avez en revanche pas besoin d'installer de driver.

3. Décompressez les fichiers de la librairie dans un répertoire de votre choix, et ouvrez une fenêtre de commande dans le répertoire où vous l'avez installée. Lancez la commande suivante pour installer les quelques dépendances qui sont nécessaires au lancement des exemples:

```
npm install
```

Une fois cette commande terminée sans erreur, vous êtes prêt pour l'exploration des exemples. Ceux-ci sont fournis dans deux exemples différents, selon l'environnement d'exécution choisi: `example_html` pour l'exécution de la librairie Yoctopuce dans un navigateur Web, ou `example_nodejs` si vous provoyez d'utiliser la librairie dans un environnement Node.js.

La manière de lancer les exemples dépend de l'environnement choisi. Vous trouverez les instructions détaillées un peu plus loin.

15.2. Petit rappel sur les fonctions asynchrones en JavaScript

JavaScript a été conçu pour éviter toute situation de *concurrence* durant l'exécution. Il n'y a jamais qu'un seul *thread* en JavaScript. Pour gérer les attentes dans les entrées/sorties, JavaScript utilise les opérations asynchrones: lorsqu'une fonction potentiellement bloquante doit être appelée, l'opération est déclenchée mais le flot d'exécution est immédiatement suspendu. Le moteur JavaScript est alors libre pour exécuter d'autres tâches, comme la gestion de l'interface utilisateur par exemple. Lorsque l'opération bloquante se termine finalement, le système relance le code en appelant une fonction de callback, en passant en paramètre le résultat de l'opération, pour permettre de continuer la tâche originale.

L'utilisation d'opérations asynchrones avec des fonctions de callback a la fâcheuse tendance de rentre le code illisible puisqu'elle découpe systématiquement le flot du code en petites fonctions de callback déconnectées les unes des autres. Heureusement, le standard ECMAScript 2015 a apporté les objets *Promise* et la syntaxe `async` / `await` pour la gestion des appels asynchrones:

- une fonction déclarée *async* encapsule automatiquement son résultat dans une promesse

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

- dans une fonction *async*, tout appel préfixé par *await* a pour effet de chaîner automatiquement la promesses retournées par la fonction appelée à une promesse de continue l'exécution de l'appelant
- tout exception durant l'exécution d'une fonction *async* déclenche le flot de traitement d'erreur de la promesse.

En clair, *async* et *await* permettent d'écrire du code TypeScript avec tous les avantages des entrées/sorties asynchrones, mais sans interrompre le flot d'écriture du code. Cela revient quasiment à une exécution multi-tâche, mais en garantissant que le passage de contrôle d'une tâche à l'autre ne se produira que là où le mot-clé *await* apparaît.

Cette librairie TypeScript utilise donc les objets *Promise* et des méthodes *async*, pour vous permettre d'utiliser la notation *await* si pratique. Et pour ne pas devoir vous poser la question pour chaque méthode de savoir si elle est asynchrone ou pas, la convention est la suivante: en principe toutes les méthodes publiques de la librairie TypeScript sont *async*, c'est-à-dire qu'elles retournent un objet Promise, sauf:

- `GetTickCount()`, parce que mesurer le temps de manière asynchrone n'a pas beaucoup de sens...
- `FindModule()`, `FirstModule()`, `nextModule()`,... parce que la détection et l'énumération des modules est faite en tâche de fond sur des structures internes qui sont gérées de manière transparente, et qu'il n'est donc pas nécessaire de faire des opérations bloquantes durant le simple parcours de ces listes de modules.

Dans la plupart des cas, le typage fort de TypeScript sera là pour vous rappeler d'utiliser `await` lors de l'appel d'une méthode asynchrone.

15.3. Contrôle de la fonction Sdi12Port

Il suffit de quelques lignes de code pour piloter un Yocto-SDI12. Voici le squelette d'un fragment de code TypeScript qui utilise la fonction Sdi12Port.

```
// En Node.js, on référence la librairie via son package NPM
// En HTML, on utiliserait plutôt un path relatif (selon l'environnement)
import { YAPI, YErrorMsg, YModule } from 'yoctolib-cjs/yocto_api_nodejs.js';
import { YSdi12Port } from 'yoctolib-cjs/yocto_sdi12port.js';

[...]
// On active l'accès aux modules locaux à travers le VirtualHub
await YAPI.RegisterHub('127.0.0.1');
[...]

// On récupère l'objet permettant d'intéragir avec le module
let sdi12port: YSdi12Port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.sdi12Port");

// Pour gérer le hot-plug, on vérifie que le module est là
if(await sdi12port.isOnline())
{
    // Utiliser sdi12port.set_sdi12Mode()
    [...]
}
```

Voyons maintenant en détail ce que font ces quelques lignes.

Import de yocto_api et yocto_sdi12port

Ces deux imports permettent d'avoir accès aux fonctions permettant de gérer les modules Yoctopuce. `yocto_api` doit toujours être inclus, et `yocto_sdi12port` est nécessaire pour gérer les modules contenant un port SDI12, comme le Yocto-SDI12. D'autres classes peuvent être utiles dans d'autres cas, comme `YModule` qui vous permet de faire une énumération de n'importe quel type de module Yoctopuce.

Pour que `yocto_api` soit correctement lié aux librairies réseau à utiliser pour établir la connexion (soit celles de Node.js, soit celles du navigateur dans le cas d'une application HTML), il faut que

vous référenciez au moins une fois dans votre projet soit la variante `yocto_api_nodejs.js`, soit `yocto_api_html.js`.

Notez que cet exemple importe la librairie au format CommonJS, le plus utilisé avec Node.JS à ce jour, mais si votre projet est construit pour utiliser les modules natifs EcmaScript, il suffit de remplace dans l'import le préfix `yoctolib-cjs` par `yoctolib-esm`.

YAPI.RegisterHub

La méthode `RegisterHub` permet d'indiquer sur quelle machine se trouvent les modules Yoctopuce, ou plus exactement la machine sur laquelle tourne le programme VirtualHub. Dans notre cas l'adresse `127.0.0.1:4444` indique la machine locale, en utilisant le port `4444` (le port standard utilisé par Yoctopuce). Vous pouvez parfaitement changer cette adresse, et mettre l'adresse d'une autre machine sur laquelle tournerait un autre VirtualHub, ou d'un YoctoHub. Si l'hôte n'est pas joignable, la fonction déclanche une exception.

Comme expliqué précédemment, il n'est pas possible d'utiliser directement `RegisterHub ("usb")` en TypeScript, car la machine virtuelle JavaScript n'a pas accès directement aux périphériques USB. Elle doit nécessairement passer par le programme VirtualHub via une connection par l'adresse 127.0.0.1.

YSdi12Port.FindSdi12Port

La méthode `FindSdi12Port` permet de retrouver un port SDI12 en fonction du numéro de série de son module hôte et de son nom de fonction. Mais vous pouvez tout aussi bien utiliser des noms logiques que vous auriez préalablement configurés. Imaginons un module Yocto-SDI12 avec le numéros de série *YSDIMK01-123456* que vous auriez appelé "*MonModule*" et dont vous auriez nommé la fonction *sdi12Port* "*MaFonction*", les cinq appels suivants seront strictement équivalents (pour autant que *MaFonction* ne soit définie qu'une fois, pour éviter toute ambiguïté):

```
sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.sdi12Port")
sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.MaFonction")
sdi12port = YSdi12Port.FindSdi12Port("MonModule.sdi12Port")
sdi12port = YSdi12Port.FindSdi12Port("MonModule.MaFonction")
sdi12port = YSdi12Port.FindSdi12Port("MaFonction")
```

`YSdi12Port.FindSdi12Port` renvoie un objet que vous pouvez ensuite utiliser à loisir pour contrôler le port SDI12.

isOnline

La méthode `isOnline()` de l'objet renvoyé par `FindSdi12Port` permet de savoir si le module correspondant est présent et en état de marche.

reset

La méthode `reset()` de l'objet returné par `YSdi12Port.FindSerialPort` vide tous les tampons du port série.

discoverSingleSensor

La méthode `discoverSingleSensor()` cherche l'adresse du capteur connecté sur le port SDI-12 et renvoie un objet avec toute les informations du capteur.

readSensor

La méthode `readSensor()` transmet la commande spécifiée sur le port SDI-12 au capteur spécifié et renvoie une liste d'objet avec toute les valeurs envoyées par le capteur.

Un exemple concret, en Node.js

Ouvrez une fenêtre de commande (un terminal, un shell...) et allez dans le répertoire **example_nodejs/Doc-GettingStarted-Yocto-SDI12** de la librairie Yoctopuce pour TypeScript. Vous y trouverez un fichier nommé `demo.ts` avec le code d'exemple ci-dessous, qui reprend les

fonctions expliquées précédemment, mais cette fois utilisées avec le décorum nécessaire à en faire un petit programme d'exemple concret.

Si le Yocto-SDI12 n'est pas branché sur la machine où fonctionne le navigateur internet, remplacez dans l'exemple l'adresse `127.0.0.1` par l'adresse IP de la machine où est branché le Yocto-SDI12 et où vous avez lancé le VirtualHub.

```
import { YAPI, YErrorMsg, YModule } from 'yoctolib-cjs/yocto_api_nodejs.js';
import { YSdi12Port } from 'yoctolib-cjs/yocto_sdi12port.js';


let singleSensor;
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 I2C interface to use
    let target: string;
    if(args[1] == "any") {
        let anySdi12 = YSdi12Port.FirstSdi12Port();
        if (anySdi12 == null) {
            console.log("No module connected (check USB cable)\n");
            process.exit(1);
        }
        let module: YModule = await anySdi12.get_module();
        target = await module.get_serialNumber();
    } else {
        target = args[1];
    }

    let sdi12Port = YSdi12Port.FindSdi12Port(target+'.sdi12Port');

    if(await sdi12Port.isOnline()) {
        console.log(target);
        singleSensor = await sdi12Port.discoverSingleSensor();
        console.log('Sensor address : ' + await singleSensor.get_sensorAddress());
        console.log('sensor SDI-12 compatibility : ' + await
singleSensor.get_sensorProtocol());
        console.log('Sensor compagny name : ' + await singleSensor.get_sensorVendor());
        console.log('Sensor model number : ' + await singleSensor.get_sensorModel());
        console.log('Sensor version : ' + await singleSensor.get_sensorVersion());
        console.log('Sensor serial number : '+ await singleSensor.get_sensorSerial());
        await YAPI.Sleep(5000 , errmsg);
        while (await sdi12Port.isOnline())
        {
            let sensorVal = await sdi12Port.readSensor(await singleSensor.get_sensorAddress
(), 'M', 5000)
            console.clear();
            console.log('Sensor address : ' + await singleSensor.get_sensorAddress());
            for (let i = 0; i < sensorVal.length; i ++)
            {
                if (await singleSensor.get_measureCount() > 1)
                {
                    console.log(await singleSensor.get_measureSymbol(i) + ' ' + sensorVal[i
] + ' ' +
                        await singleSensor.get_measureUnit(i) + ' ' + await
singleSensor.get_measureDescription(i));
                }
                else
                {
                    console.log(sensorVal[i]);
                }
            }
            await YAPI.Sleep(5000, errmsg);
        }
    } else {
        console.log("Module not connected (check identification and USB cable)\n");
    }
```

```
    await YAPI.FreeAPI();
}

if(process.argv.length < 3) {
    console.log("usage: node demo.js <serial_number>");
    console.log("       node demo.js <logical_name>");
    console.log("       node demo.js any            (use any discovered device)");
} else {
    startDemo(process.argv.slice(process.argv.length - 2));
}
```

Comme décrit au début de ce chapitre, vous devez avoir installé le complateur TypeScript sur votre machine pour essayer ces exemples, et installé les dépendances de la librairie TypeScript. Si vous l'avez fait, vous pouvez maintenant taper la commande suivantes dans le répertoire de l'exemple lui-même, pour finaliser la résolution de ses dépendances:

```
npm install
```

Vous êtes maintenant prêt pour lancer le code d'exemple dans Node.js. La manière la plus simple de le faire est d'utiliser la commande suivante, en remplaçant les [...] par les arguments que vous voulez passer au programme:

```
npm run demo [...]
```

Cette commande, définie dans le fichier `package.json`, a pour effet de compiler le code source TypeScript à l'aide de la simple commande `tsc`, puis de lancer le code compilé dans Node.js.

La compilation utilise les paramètres spécifiés dans le fichier `tsconfig.json`, et produit

- un fichier JavaScript `demo.js`, que Node.js pourra exécuter
- un fichier de debug `demo.js.map`, qui permettra le cas échéant à Node.js de signaler les erreurs en référançant leur origine dans le fichier d'origine en TypeScript.

Notez que le fichier `package.json` de nos exemples référence directement la version locale de la librairie par un path relatif, pour éviter de dupliquer la librairie dans chaque exemple. Bien sur, pour votre application de production, vous pourrez utiliser le package directement depuis le repository npm en l'ajoutant à votre projet à l'aide de la commande:

```
npm install yoctolib-cjs
```

Le même exemple, mais dans un navigateur

Si vous voulez voir comment utiliser la librairie dans un navigateur plutôt que dans Node.js, changez de répertoire et allez dans **example_html/Doc-GettingStarted-Yocto-SDI12**. Vous y trouverez un fichier html `app.html`, et un fichier TypeScript `app.ts` similaire au code ci-dessus, mais avec quelques variantes pour permettre une interaction à travers la page HTML plutôt que sur la console JavaScript.

Aucune installation n'est nécessaire pout utiliser cet exemple HTML, puisqu'il référence la librairie TypeScript via un chemin relatif. Par contre, pour que le navigateur puisse exécuter le code, il faut que la page HTML soit publié par un serveur Web. Nous fournissons un petit serveur de test pour cet usage, que vous pouvez lancer avec la commande:

```
npm run app-server
```

Cette commande va compiler le code d'exemple TypeScript, le mettre à disposition via un serveur HTTP sur le port 3000 et ouvrir un navigateur sur cet exemple. Si vous modifiez le code d'exemple, il sera automatiquement recompilé et il vous suffira de recharger la page sur le navigateur pour retester.

Comme pour l'exemple Node.js, la compilation produit un fichier `.js.map` qui permet de debugger dans le navigateur directement sur le fichier source TypeScript. Notez qu'au moment où cette documentation est rédigée, le debug en format source dans le navigateur fonctionne pour les browsers basés sur Chromium, mais pas encore dans Firefox.

15.4. Contrôle de la partie module

Chaque module peut-être contrôlé d'une manière similaire, vous trouverez ci dessous un simple programme d'exemple affichant les principaux paramètres d'un module et permettant d'activer la balise de localisation.

```
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));
}
```

Chaque propriété `xxx` du module peut être lue grâce à une méthode du type `get_xxxx()`, et les propriétés qui se sont pas en lecture seule peuvent être modifiées à l'aide de la méthode `set_xxx()` Pour plus de détails concernant ces méthodes utilisées, reportez-vous aux chapitre API

Modifications des réglages du module

Lorsque que vous souhaitez modifier les réglages d'un module, il suffit d'appeler la méthode `set_xxx()` correspondante, cependant cette modification n'a lieu que dans la mémoire vive du module: si le module redémarre, les modifications seront perdues. Pour qu'elle soient mémorisées de manière persistante, il est nécessaire de demander au module de sauvegarder sa configuration courante dans sa mémoire non volatile. Pour cela il faut utiliser la méthode `saveToFlash()`. Inversement il est possible de forcer le module à oublier ses réglages courants en utilisant la méthode `revertFromFlash()`. Ce petit exemple ci-dessous vous permet changer le nom logique d'un 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));
}
```

Attention, le nombre de cycle d'écriture de la mémoire non volatile du module est limité. Passé cette limite plus rien ne garantit de que la sauvegarde des réglages se passera correctement. Cette limite, liée à la technologie employé par le micro-processeur du module se situe aux alentour de 100000 cycles. Pour résumer vous ne pouvez employer la méthode `saveToFlash()` que 100000 fois au cours de la vie du module. Veillez donc à ne pas appeler cette méthode depuis l'intérieur d'une boucle.

Énumération des modules

Obtenir la liste des modules connectés se fait à l'aide de la méthode `YModule.FirstModule()` qui renvoie le premier module trouvé, il suffit ensuite d'appeler la méthode `nextModule()` de cet objet pour trouver les modules suivants, et ce tant que la réponse n'est pas un `null`. Ci-dessous un petit exemple listant les module connectés

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

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

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

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

        let module = YModule.FirstModule();
```

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

startDemo();
```

15.5. Gestion des erreurs

Lorsque vous implémentez un programme qui doit interagir avec des modules USB, vous ne pouvez pas faire abstraction de la gestion des erreurs. Il y aura forcément une occasion où un utilisateur aura débranché le périphérique, soit avant de lancer le programme, soit même en pleine opération. La librairie Yoctopuce est prévue pour vous aider à supporter ce genre de comportements, mais votre code doit néanmoins être fait pour se comporter au mieux pour interpréter les erreurs signalées par la librairie.

La manière la plus simple de contourner le problème est celle que nous avons employé pour les petits exemples précédents de ce chapitre: avant d'accéder à un module, on vérifie qu'il est en ligne avec la méthode `isOnline()` et on suppose ensuite qu'il va y rester pendant la fraction de seconde nécessaire à exécuter les lignes de code suivantes. Ce n'est pas parfait, mais ça peut suffire dans certains cas. Il faut toutefois être conscient qu'on ne peut pas totalement exclure une erreur se produisant après le `isOnline()`, qui pourrait faire planter le programme. La seule manière de l'éviter est d'implémenter une des deux techniques de gestion des erreurs décrites ci-dessous.

La méthode recommandée par la plupart des langages de programmation pour la gestion des erreurs imprévisibles est l'utilisation d'exceptions. C'est le comportement par défaut de la librairie Yoctopuce. Si une erreur se produit alors qu'on essaie d'accéder à un module, la librairie va lancer une exception. Dans ce cas, de trois choses l'une:

- Si votre code attrape l'exception au vol et la gère, et tout se passe bien.
- Si votre programme tourne dans le debugger, vous pourrez relativement facilement déterminer où le problème s'est produit, et voir le message explicatif lié à l'exception.
- Sinon... l'exception va crasher votre programme, boum!

Comme cette dernière situation n'est pas la plus souhaitable, la librairie Yoctopuce offre une autre alternative pour la gestion des erreurs, permettant de faire un programme robuste sans devoir attraper les exceptions à chaque ligne de code. Il suffit d'appeler la fonction `YAPI.DisableExceptions()` pour commuter la librairie dans un mode où les exceptions de chaque fonction sont systématiquement remplacées par des valeurs de retour particulières, qui peuvent être testées par l'appelant lorsque c'est pertinent. Le nom de la valeur de retour en cas d'erreur pour chaque fonction est systématiquement documenté dans la référence de la librairie. Il suit toujours la même logique: une méthode `get_state()` retournera une valeur `NomDeClasse.STATE_INVALID`, une méthode `get_currentValue` retournera une valeur `NomDeClasse.CURRENTVALUE_INVALID`, etc. Dans tous les cas, la valeur retournée sera du type attendu, et ne sera pas un pointeur nul qui risquerait de faire crasher votre programme. Au pire, si vous affichez la valeur sans la tester, elle sera hors du cadre attendu pour la valeur retournée. Dans le cas de fonctions qui ne retournent à priori pas d'information, la valeur de retour sera `YAPI.SUCCESS` si tout va bien, et un code d'erreur différent en cas d'échec.

Quand vous travaillez sans les exceptions, il est possible d'obtenir un code d'erreur et un message expliquant l'origine de l'erreur en le demandant à l'objet qui a retourné une erreur à l'aide des méthodes `errType()` et `errMessage()`. Ce sont les même informations qui auraient été associées à l'exception si elles avaient été actives.

16. Utilisation du Yocto-SDI12 en JavaScript / EcmaScript

EcmaScript est le nom officiel de la version standardisée du langage de programmation communément appelé JavaScript. Cette librairie de programmation Yoctopuce utilise les nouvelles fonctionnalités introduites dans la version EcmaScript 2017. La librairie porte ainsi le nom *Librairie pour JavaScript / EcmaScript 2017*, afin de la différentier de la précédente *Librairie pour JavaScript* qu'elle remplace.

Cette librairie permet d'accéder aux modules Yoctopuce depuis tous les environnements JavaScript modernes. Elle fonctionne aussi bien depuis un navigateur internet que dans un environnement Node.js. La librairie détecte automatiquement à l'initialisation si le contexte d'utilisation est un browser ou une machine virtuelle Node.js, et utilise les librairies systèmes les plus appropriées en conséquence.

Les communications asynchrones avec les modules sont gérées dans toute la librairie à l'aide d'objets *Promise*, en utilisant la nouvelle syntaxe EcmaScript 2017 `async` / `await` non bloquante pour la gestion des entrées/sorties asynchrones (voir ci-dessous). Cette syntaxe est désormais disponible sans autres dans la plupart des moteurs JavaScript: il n'est plus nécessaire de transpiler le code avec Babel ou `jspm`. Voici la version minimum requise de vos moteurs JavaScript préférés, tous disponibles au téléchargement:

- 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

Si vous avez besoin de la compatibilité avec des anciennes versions, vous pouvez toujours utiliser Babel pour transpiler votre code et la libriairie vers un standard antérieur de JavaScript, comme décrit un peu plus bas.

Nous ne recommendons plus l'utilisation de `jspm` dès lors que `async` / `await` sont standardisés.

16.1. Fonctions bloquantes et fonctions asynchrones en JavaScript

JavaScript a été conçu pour éviter toute situation de *concurrence* durant l'exécution. Il n'y a jamais qu'un seul *thread* en JavaScript. Cela signifie que si un programme effectue une attente active durant une communication réseau, par exemple pour lire un capteur, le programme entier se trouve bloqué. Dans un navigateur, cela peut se traduire par un blocage complet de l'interface utilisateur. C'est pourquoi l'utilisation de fonctions d'entrée/sortie bloquantes en JavaScript est sévèrement découragée de nos jours, et les API bloquantes se font toutes déclarer *deprecated*.

Plutôt que d'utiliser des *threads* parallèles, JavaScript utilise les opérations asynchrones pour gérer les attentes dans les entrées/sorties: lorsqu'une fonction potentiellement bloquante doit être appelée, l'opération est uniquement déclenchée mais le flot d'exécution est immédiatement terminé. La moteur JavaScript est alors libre pour exécuter d'autres tâches, comme la gestion de l'interface utilisateur par exemple. Lorsque l'opération bloquante se termine finalement, le système relance le code en appelant une fonction de callback, en passant en paramètre le résultat de l'opération, pour permettre de continuer la tâche originale.

Lorsqu'on les utilises avec des simples fonctions de callback, comme c'est fait quasi systématiquement dans les librairies Node.js, les opérations asynchrones ont la fâcheuse tendance de rentre le code illisible puisqu'elles découpent systématiquement le flot du code en petites fonctions de callback déconnectées les unes des autres. Heureusement, de nouvelles idées sont apparues récemment pour améliorer la situation. En particulier, l'utilisation d'objets *Promise* pour travailler avec les opérations asynchrones aide beaucoup. N'importe quelle fonction qui effectue une opération potentiellement longue peut retourner une *promesse* de se terminer, et cet objet *Promise* peut être utilisé par l'appelant pour chaîner d'autres opérations en un flot d'exécution. La classe *Promise* fait partie du standard EcmaScript 2015.

Les objets *Promise* sont utiles, mais ce qui les rend vraiment pratique est la nouvelle syntaxe `async` / `await` pour la gestion des appels asynchrones:

- une fonction déclarée *async* encapsule automatiquement son résultat dans une promesse
- dans une fonction *async*, tout appel préfixé par *await* a pour effet de chaîner automatiquement la promesses retournées par la fonction appelée à une promesse de continue l'exécution de l'appelant
- tout exception durant l'exécution d'une fonction *async* déclenche le flot de traitrment d'erreur de la promesse.

En clair, *async* et *await* permettent d'écrire du code EcmaScript avec tous les avantages des entrées/sorties asynchrones, mais sans interrompre le flot d'écriture du code. Cela revient quasiment à une exécution multi-tâche, mais en garantissant que le passage de contrôle d'une tâche à l'autre ne se produira que là où le mot-clé *await* apparaît.

Nous avons donc décidé d'écrire cette nouvelle librairie EcmaScript en utilisant les objets *Promise* et des fonctions *async*, pour vous permettre d'utiliser la notation *await* si pratique. Et pour ne pas devoir vous poser la question pour chaque méthode de savoir si elle est asynchrone ou pas, la convention est la suivante: **toutes les méthodes publiques** de la librairie EcmaScript **sont *async***, c'est-à-dire qu'elles retournent un objet Promise, **sauf**:

- `GetTickCount()`, parce que mesurer le temps de manière asynchrone n'a pas beaucoup de sens...
- `FindModule(), FirstModule(), nextModule()`, ... parce que la détection et l'énumération des modules est faite en tâche de fond sur des structures internes qui sont gérées de manière transparente, et qu'il n'est donc pas nécessaire de faire des opérations bloquantes durant le simple parcours de ces listes de modules.

16.2. Utiliser la librairie Yoctopuce pour JavaScript / EcmaScript 2017

JavaScript fait partie de ces langages qui ne vous permettront pas d'accéder directement aux couches matérielles de votre ordinateur. C'est pourquoi si vous désirez travailler avec des modules USB branchés par USB, vous devrez faire tourner la passerelle de Yoctopuce appelée VirtualHub sur la machine à laquelle sont branchés les modules.

Connectez vous sur le site de Yoctopuce et téléchargez les éléments suivants:

- La librairie de programmation pour Javascript / EcmaScript 2017[1]
- VirtualHub[2] pour Windows, macOS ou Linux selon l'OS que vous utilisez

Décompressez les fichiers de la librairie dans un répertoire de votre choix, branchez vos modules et lancez le programme VirtualHub. Vous n'avez pas besoin d'installer de driver.

Utiliser la librairie Yoctopuce officielle pour node.js

Commencez par installer sur votre machine de développement la version actuelle de Node.js (7.6 ou plus récente), C'est très simple. Vous pouvez l'obtenir sur le site officiel: http://nodejs.org. Assurez vous de l'installer entièrement, y compris npm, et de l'ajouter à votre system path.

Vous pouvez ensuite prendre l'exemple de votre choix dans le répertoire `example_nodejs` (par exemple `example_nodejs/Doc-Inventory`). Allez dans ce répertoire. Vous y trouverez un fichier décrivant l'application (`package.json`) et le code source de l'application (`demo.js`). Pour charger automatiquement et configurer les librairies nécessaires à l'exemple, tapez simplement:

```
npm install
```

Une fois que c'est fait, vous pouvez directement lancer le code de l'application:

```
node demo.js
```

Utiliser une copie locale de la librairie Yoctopuce avec node.js

Si pour une raison ou une autre vous devez faire des modifications au code de la librairie, vous pouvez facilement configurer votre projet pour utiliser le code source de la librairie qui se trouve dans le répertoire `lib/` plutôt que le package npm officiel. Pour cela, lancez simplement la commande suivante dans le répertoire de votre projet:

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

Utiliser la librairie Yoctopuce dans un navigateur (HTML)

Pour les exemples HTML, c'est encore plus simple: il n'y a rien à installer. Chaque exemple est un simple fichier HTML que vous pouvez ouvrir directement avec un navigateur pour l'essayer. L'inclusion de la librairie Yoctopuce ne demande rien de plus qu'un simple tag HTML `<script>`.

Utiliser la librairie Yoctopuce avec des anciennes version de JavaScript

Si vous avez besoin d'utiliser cette librairie avec des moteurs JavaScript plus anciens, vous pouvez utiliser Babel[3] pour transpiler votre code et la librairie dans une version antérieure du langage. Pour installer Babel avec les réglages usuels, tapez:

[1] www.yoctopuce.com/FR/libraries.php
[2] www.yoctopuce.com/FR/virtualhub.php
[3] http://babeljs.io

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

Normalement vous demanderez à Babel de poser les fichiers transpilés dans un autre répertoire, nommé `comopat` par exemple. Pour ce faire, utilisez par exemple les commandes suivantes:

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

Bien que ces outils de transpilation soient basés sur node.js, ils fonctionnent en réalité pour traduire n'importe quel type de fichier JavaScript, y compris du code destiné à fonctionner dans un navigateur. La seule chose qui ne peut pas être faite aussi facilement est la transpilation de sciptes codés en dure à l'intérieur même d'une page HTML. Il vous faudra donc sortir ce code dans un fichier `.js` externe si il utiliser la syntaxe EcmaScript 2017, afin de le transpiler séparément avec Babel.

Babel dipose de nombreuses fonctionnalités intéressantes, comme un mode de surveillance qui traduite automatiquement au vol vos fichiers dès qu'il détecte qu'un fichier source a changé. Consultez les détails dans la documentation de Babel.

Compatibilité avec l'ancienne librairie JavaScript

Cette nouvelle librairie n'est pas compatible avec l'ancienne librairie JavaScript, car il n'existe pas de possibilité d'implémenter l'ancienne API bloquante sur la base d'une API asynchrone. Toutefois, les noms des méthodes sont les mêmes, et l'ancien code source synchrone peut facilement être rendu asynchrone simplement en ajoutant le mot-clé `await` devant les appels de méthode. Remplacez par exemple:

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

par

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

Mis à part quelques exceptions, la plupart des méthodes redondantes `XXX_async` ont été supprimées, car elles auraient introduit de la confusion sur la manière correcte de gérer les appels asynchrones. Si toutefois vous avez besoin d'appeler un callback explicitement, il est très facile de faire appeler une fonction de callback à la résolution d'une méthode *async*, en utilisant l'objet Promise retourné. Par exemple, vous pouvez réécrire:

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

par

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

Si vous portez une application vers la nouvelle librairie, vous pourriez être amené à désirer des méthodes synchrones similaires à l'ancienne librairie (sans objet Promise), quitte à ce qu'elles retournent la dernière valeur reçue du capteur telle que stockée en cache, puisqu'il n'est pas possible de faire des communications bloquantes. Pour cela, la nouvelle librairie introduit un nouveau type de classes appelés *proxys synchrones*. Un proxy synchrone est un objet qui reflète la dernière value connue d'un objet d'interface, mais peut être accédé à l'aide de fonctions synchrones habituelles. Par exemple, plutôt que d'utiliser:

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

...
```

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

on peut utiliser:

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

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

Ce dernier appel asynchrone peut aussi être formulé comme:

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

16.3. Contrôle de la fonction Sdi12Port

Il suffit de quelques lignes de code pour piloter un Yocto-SDI12. Voici le squelette d'un fragment de code JavaScript qui utilise la fonction Sdi12Port.

```
// En Node.js, on utilise la fonction require()
// En HTML, on utiliserait &lt;script src="..."&gt;
require('yoctolib-es2017/yocto_api.js');
require('yoctolib-es2017/yocto_sdi12port.js');

[...]
// On active l'accès aux modules locaux à travers le VirtualHub
await YAPI.RegisterHub('127.0.0.1');
[...]

// On récupère l'objet permettant d'intéragir avec le module
let sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.sdi12Port");

// Pour gérer le hot-plug, on vérifie que le module est là
if(await sdi12port.isOnline())
{
    // Utiliser sdi12port.set_sdi12Mode()
    [...]
}
```

Voyons maintenant en détail ce que font ces quelques lignes.

Require de yocto_api et yocto_sdi12port

Ces deux imports permettent d'avoir accès aux fonctions permettant de gérer les modules Yoctopuce. `yocto_api` doit toujours être inclus, `yocto_sdi12port` est nécessaire pour gérer les modules contenant un port SDI12, comme le Yocto-SDI12. D'autres classes peuvent être utiles dans d'autres cas, comme `YModule` qui vous permet de faire une énumération de n'importe quel type de module Yoctopuce.

YAPI.RegisterHub

La méthode `RegisterHub` permet d'indiquer sur quelle machine se trouvent les modules Yoctopuce, ou plus exactement la machine sur laquelle tourne le programme VirtualHub. Dans notre cas l'adresse `127.0.0.1:4444` indique la machine locale, en utilisant le port `4444` (le port standard utilisé par Yoctopuce). Vous pouvez parfaitement changer cette adresse, et mettre l'adresse d'une autre machine sur laquelle tournerait un autre VirtualHub, ou d'un YoctoHub. Si l'hôte n'est pas joignable, la fonction déclanche une exception.

YSdi12Port.FindSdi12Port

La méthode `FindSdi12Port` permet de retrouver un port SDI12 en fonction du numéro de série de son module hôte et de son nom de fonction. Mais vous pouvez tout aussi bien utiliser des noms

logiques que vous auriez préalablement configurés. Imaginons un module Yocto-SDI12 avec le numéros de série *YSDIMK01-123456* que vous auriez appelé "*MonModule*" et dont vous auriez nommé la fonction *sdi12Port* "*MaFonction*", les cinq appels suivants seront strictement équivalents (pour autant que *MaFonction* ne soit définie qu'une fois, pour éviter toute ambiguïté):

```
sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.sdi12Port")
sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.MaFonction")
sdi12port = YSdi12Port.FindSdi12Port("MonModule.sdi12Port")
sdi12port = YSdi12Port.FindSdi12Port("MonModule.MaFonction")
sdi12port = YSdi12Port.FindSdi12Port("MaFonction")
```

`YSdi12Port.FindSdi12Port` renvoie un objet que vous pouvez ensuite utiliser à loisir pour contrôler le port SDI12.

isOnline

La méthode `isOnline()` de l'objet renvoyé par `FindSdi12Port` permet de savoir si le module correspondant est présent et en état de marche.

reset

La méthode `reset()` de l'objet returné par `YSdi12Port.FindSerialPort` vide tous les tampons du port série.

discoverSingleSensor

La méthode `discoverSingleSensor()` cherche l'adresse du capteur connecté sur le port SDI-12 et renvoie un objet avec toute les informations du capteur.

readSensor

La méthode `readSensor()` transmet la commande spécifiée sur le port SDI-12 au capteur spécifié et renvoie une liste d'objet avec toute les valeurs envoyées par le capteur.

Un exemple concret, en Node.js

Ouvrez une fenêtre de commande (un terminal, un shell...) et allez dans le répertoire **example_nodejs/Doc-GettingStarted-Yocto-SDI12** de la librairie Yoctopuce pour JavaScript / EcmaScript 2017. Vous y trouverez un fichier nommé `demo.js` avec le code d'exemple ci-dessous, qui reprend les fonctions expliquées précédemment, mais cette fois utilisées avec le décorum nécessaire à en faire un petit programme d'exemple concret.

Si le Yocto-SDI12 n'est pas branché sur la machine où fonctionne le navigateur internet, remplacez dans l'exemple l'adresse `127.0.0.1` par l'adresse IP de la machine où est branché le Yocto-SDI12 et où vous avez lancé le VirtualHub.

```
"use strict";

require('yoctolib-es2017/yocto_api.js');
require('yoctolib-es2017/yocto_sdi12port.js');

let sdi12Port;
let singleSensor;

async function startDemo() {
    const readline = YAPI._nodeRequire('readline');
    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;
    }

        // by default use any connected module suitable for the demo    //
    let anySdi12 = YSdi12Port.FirstSdi12Port();
    if(anySdi12) {
```

```
        let module = await anySdi12.module();
        let target = await module.get_serialNumber();
        console.log('Using device : ' + target);
        sdi12Port = YSdi12Port.FindSdi12Port(target + ".sdi12Port");
    }
    if (await sdi12Port.isOnline()) {
        singleSensor = await sdi12Port.discoverSingleSensor();
        console.log('Sensor address :' + await singleSensor.get_sensorAddress());
        console.log('sensor SDI-12 compatibility : ' + await
singleSensor.get_sensorProtocol());
        console.log('Sensor compagny name : ' + await singleSensor.get_sensorVendor());
        console.log('Sensor model number : ' + await singleSensor.get_sensorModel());
        console.log('Sensor version : ' + await singleSensor.get_sensorVersion());
        console.log('Sensor serial number : '+ await singleSensor.get_sensorSerial());
        await YAPI.Sleep(5000, errmsg);
        await refresh();
    }
}

async function refresh() {

    if (await sdi12Port.isOnline()) {
        let sensorVal = await sdi12Port.readSensor(await singleSensor.get_sensorAddress(),
'M', 5000)
        console.clear();
        console.log('Sensor address : ' + await singleSensor.get_sensorAddress());
        for (let i = 0; i < sensorVal.length; i ++)
        {
            if (await singleSensor.get_measureCount() > 1)
            {
                 console.log(await singleSensor.get_measureSymbol(i) + ' ' + sensorVal[i] +
' ' +
                     await singleSensor.get_measureUnit(i) + ' ' + await
singleSensor.get_measureDescription(i));
            }
            else
            {
                console.log(sensorVal[i]);
            }
        }
    } else {
        console.log('Module not connected');
    }
    setTimeout(refresh, 5000);

}

startDemo();
```

Comme décrit au début de ce chapitre, vous devez avoir installé Node.js v7.6 ou suivant pour essayer ces exemples. Si vous l'avez fait, vous pouvez maintenant taper les deux commandes suivantes pour télécharger automatiquement les librairies dont cet exemple dépend:

```
npm install
```

Une fois terminé, vous pouvez lancer votre code d'exemple dans Node.js avec la commande suivante, en remplaçant les [...] par les arguments que vous voulez passer au programme:

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

Le même exemple, mais dans un navigateur

Si vous voulez voir comment utiliser la librairie dans un navigateur plutôt que dans Node.js, changez de répertoire et allez dans **example_html/Doc-GettingStarted-Yocto-SDI12**. Vous y trouverez un fichier html, avec une section JavaScript similaire au code précédent, mais avec quelques variantes pour permettre une interaction à travers la page HTML plutôt que sur la console JavaScript

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

```
  <title>Hello World</title>
  <script src="../../lib/yocto_api.js"></script>
  <script src="../../lib/yocto_sdi12port.js"></script>
  <script>
    let sdi12Port;

    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 anySdi12 = YSdi12Port.FirstSdi12Port();
        if(anySdi12) {
          let module = await anySdi12.module();
          serial = await module.get_serialNumber();
          document.getElementById('serial').value = serial;
        }
      }
      sdi12Port = YSdi12Port.FindSdi12Port(serial+'.sdi12Port');
      if(await sdi12Port.isOnline()) {

        let singleSensor = await sdi12Port.discoverSingleSensor();

        document.getElementById('addr').value = await singleSensor.get_sensorAddress();
        document.getElementById('proto').value = await singleSensor.get_sensorProtocol();
        document.getElementById('vendor').value = await singleSensor.get_sensorVendor();
        document.getElementById('model').value = await singleSensor.get_sensorModel();
        document.getElementById('ver').value = await singleSensor.get_sensorVersion();
        document.getElementById('numb').value = await singleSensor.get_sensorSerial();
        let valSensor = await sdi12Port.readSensor((await singleSensor.get_sensorAddress
()).toString(),"M",5000);
        let result = '';
        for (var i = 0; i < valSensor.length ; i++) {
         if (await singleSensor.get_measureCount() > 1){
            result += (await singleSensor.get_measureSymbol(i)) + ' : ' + valSensor
[i].toString() + ' ' +
            (await singleSensor.get_measureUnit(i)) + ' ' + (await
singleSensor.get_measureDescription(i)) + '\n';
          }
          else
          {
            result += valSensor[i].toString() +'\n';
          }
        }
        document.getElementById('val').value = result;
      }
      setTimeout(refresh, 5000);
    }

    startDemo();
  </script>
</head>
<body>
Module to use: <input id='serial' readonly><br>
Sensor address : <input id='addr' readonly><br>
Sensor SDI-12 compatibility : <input id='proto' readonly><br>
Sensor company name : <input id='vendor' readonly><br>
Sensor model number : <input id='model' readonly><br>
Sensor version : <input id='ver' readonly><br>
Sensor serial number : <input id='numb' readonly><br>
<textarea id= 'val' rows="6" cols="60" readonly ></textarea>
</body>
</html>
```

Aucune installation n'est nécessaire pout utiliser cet exemple, il suffit d'ouvrir la page HTML avec un navigateur web.

16.4. Contrôle de la partie module

Chaque module peut-être contrôlé d'une manière similaire, vous trouverez ci dessous un simple programme d'exemple affichant les principaux paramètres d'un module et permettant d'activer la balise de localisation.

```
"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));
}
```

Chaque propriété `xxx` du module peut être lue grâce à une méthode du type `get_xxxx()`, et les propriétés qui se sont pas en lecture seule peuvent être modifiées à l'aide de la méthode `set_xxx()` Pour plus de détails concernant ces fonctions utilisées, reportez-vous aux chapitre API

Modifications des réglages du module

Lorsque que vous souhaitez modifier les réglages d'un module, il suffit d'appeler la fonction `set_xxx()` correspondante, cependant cette modification n'a lieu que dans la mémoire vive du module: si le module redémarre, les modifications seront perdues. Pour qu'elle soient mémorisées de manière persistante, il est nécessaire de demander au module de sauvegarder sa configuration courante dans sa mémoire non volatile. Pour cela il faut utiliser la méthode `saveToFlash()`. Inversement il est possible de forcer le module à oublier ses réglages courants en utilisant la méthode `revertFromFlash()`. Ce petit exemple ci-dessous vous permet changer le nom logique d'un 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));
}
```

Attention, le nombre de cycle d'écriture de la mémoire non volatile du module est limité. Passé cette limite plus rien ne garantit de que la sauvegarde des réglages se passera correctement. Cette limite, liée à la technologie employé par le micro-processeur du module se situe aux alentour de 100000 cycles. Pour résumer vous ne pouvez employer la fonction `saveToFlash()` que 100000 fois au cours de la vie du module. Veillez donc à ne pas appeler cette fonction depuis l'intérieur d'une boucle.

Énumération des modules

Obtenir la liste des modules connectés se fait à l'aide de la fonction `YModule.FirstModule()` qui renvoie le premier module trouvé, il suffit ensuite d'appeler la fonction `nextModule()` de cet objet pour trouver les modules suivants, et ce tant que la réponse n'est pas un `null`. Ci-dessous un petit exemple listant les module connectés

```
"use strict";

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

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

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

async function refresh()
{
```

```
    try {
        let errmsg = new YErrorMsg();
        await YAPI.UpdateDeviceList(errmsg);

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

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

16.5. Gestion des erreurs

Lorsque vous implémentez un programme qui doit interagir avec des modules USB, vous ne pouvez pas faire abstraction de la gestion des erreurs. Il y aura forcément une occasion où un utilisateur aura débranché le périphérique, soit avant de lancer le programme, soit même en pleine opération. La librairie Yoctopuce est prévue pour vous aider à supporter ce genre de comportements, mais votre code doit néanmoins être fait pour se comporter au mieux pour interpréter les erreurs signalées par la librairie.

La manière la plus simple de contourner le problème est celle que nous avons employé pour les petits exemples précédents de ce chapitre: avant d'accéder à un module, on vérifie qu'il est en ligne avec la méthode `isOnline()` et on suppose ensuite qu'il va y rester pendant la fraction de seconde nécessaire à exécuter les lignes de code suivantes. Ce n'est pas parfait, mais ça peut suffire dans certains cas. Il faut toutefois être conscient qu'on ne peut pas totalement exclure une erreur se produisant après le `isOnline()`, qui pourrait faire planter le programme. La seule manière de l'éviter est d'implémenter une des deux techniques de gestion des erreurs décrites ci-dessous.

La méthode recommandée par la plupart des langages de programmation pour la gestion des erreurs imprévisibles est l'utilisation d'exceptions. C'est le comportement par défaut de la librairie Yoctopuce. Si une erreur se produit alors qu'on essaie d'accéder à un module, la librairie va lancer une exception. Dans ce cas, de trois choses l'une:

- Si votre code attrape l'exception au vol et la gère, et tout se passe bien.
- Si votre programme tourne dans le debugger, vous pourrez relativement facilement déterminer où le problème s'est produit, et voir le message explicatif lié à l'exception.
- Sinon... l'exception va crasher votre programme, boum!

Comme cette dernière situation n'est pas la plus souhaitable, la librairie Yoctopuce offre une autre alternative pour la gestion des erreurs, permettant de faire un programme robuste sans devoir attraper les exceptions à chaque ligne de code. Il suffit d'appeler la fonction `YAPI.DisableExceptions()` pour commuter la librairie dans un mode où les exceptions de chaque fonction sont systématiquement remplacées par des valeurs de retour particulières, qui peuvent être testées par l'appelant lorsque c'est pertinent. Le nom de la valeur de retour en cas d'erreur pour chaque fonction est systématiquement documenté dans la référence de la librairie. Il suit toujours la même logique: une méthode `get_state()` retournera une valeur `NomDeClasse.STATE_INVALID`, une méthode `get_currentValue` retournera une valeur `NomDeClasse.CURRENTVALUE_INVALID`, etc. Dans tous les cas, la valeur retournée sera du type attendu, et ne sera pas un pointeur nul qui risquerait de faire crasher votre programme. Au pire, si vous affichez la valeur sans la tester, elle sera hors du cadre attendu pour la valeur retournée.

Dans le cas de fonctions qui ne retournent à priori pas d'information, la valeur de retour sera `YAPI.SUCCESS` si tout va bien, et un code d'erreur différent en cas d'échec.

Quand vous travaillez sans les exceptions, il est possible d'obtenir un code d'erreur et un message expliquant l'origine de l'erreur en le demandant à l'objet qui a retourné une erreur à l'aide des méthodes `errType()` et `errMessage()`. Ce sont les même informations qui auraient été associées à l'exception si elles avaient été actives.

17. Utilisation du Yocto-SDI12 en PHP

PHP est, tout comme Javascript, un langage assez atypique lorsqu'il s'agit de discuter avec du hardware. Néanmoins, utiliser PHP avec des modules Yoctopuce offre l'opportunité de construire très facilement des sites web capables d'interagir avec leur environnement physique, ce qui n'est pas donné à tous les serveurs web. Cette technique trouve une application directe dans la domotique: quelques modules Yoctopuce, un serveur PHP et vous pourrez interagir avec votre maison depuis n'importe ou dans le monde. Pour autant que vous ayez une connexion internet.

PHP fait lui aussi partie de ces langages qui ne vous permettront pas d'accéder directement aux couches matérielles de votre ordinateur. C'est pourquoi vous devrez faire tourner VirtualHub sur la machine à laquelle sont branchés les modules.

Pour démarrer vos essais en PHP, vous allez avoir besoin d'un serveur PHP 7.1 ou plus récent [1] de préférence en local sur votre machine. Si vous souhaiter utiliser celui qui se trouve chez votre provider internet, c'est possible, mais vous devrez probablement configurer votre routeur ADSL pour qu'il accepte et forwarde les requêtes TCP sur le port 4444.

17.1. Préparation

Connectez vous sur le site de Yoctopuce et téléchargez les éléments suivants:

- La librairie de programmation pour PHP[2]
- VirtualHub[3] pour Windows, macOS ou Linux selon l'OS que vous utilisez

Notre librairie PHP est basée sur PHP 8.x. C'est-à-dire que notre librairie fonctionne parfaitement avec n'importe quelle version de PHP actuellement encore supportée. Toutefois, afin de ne pas abandonner nos clients qui ont des installations plus anciennes, nous maintenons une version compatible avec PHP 7.1. qui date de 2016.

Par ailleurs, nous proposons également une version de la librairie qui suit les recommandations PSR. Pour simplifier, cette version est de même code que la version php8 mais chaque classe est stockée dans un fichier séparé. De plus, cette version utilise un namespace `Yoctopuce\YoctoAPI`. Ces changements rendent notre librairie beaucoup plus facilement utilisable avec des installations qui utilisent l'autoload.

Notez que les exemples de la documentation n'utilisent pas la version PSR.

[1] Quelques serveurs PHP gratuits: easyPHP pour Windows, MAMP pour macOs
[2] www.yoctopuce.com/FR/libraries.php
[3] www.yoctopuce.com/FR/virtualhub.php

Dans l'archive de la librairie, il y a donc trois sous-répertoire :

- php7
- php8
- phpPSR

Choisissez le bon répertoire en fonction de la version de la librairie que vous souhaitez utiliser, décompressez les fichiers de ce répertoire dans un répertoire de votre choix accessible à votre serveur web, branchez vos modules, lancez VirtualHub, et vous pouvez commencer vos premiers tests. Vous n'avez pas besoin d'installer de driver.

17.2. Contrôle de la fonction Sdi12Port

Il suffit de quelques lignes de code pour piloter un Yocto-SDI12. Voici le squelette d'un fragment de code PHP qui utilise la fonction Sdi12Port.

```
include('yocto_api.php');
include('yocto_sdi12port.php');

[...]
// On active l'accès aux modules locaux à travers le VirtualHub
YAPI::RegisterHub('http://127.0.0.1:4444/',$errmsg);
[...]

// On récupère l'objet permettant d'intéragir avec le module
$sdi12port = YSdi12Port::FindSdi12Port("YSDIMK01-123456.sdi12Port");

// Pour gérer le hot-plug, on vérifie que le module est là
if($sdi12port->isOnline())
{
    // Utiliser sdi12port->set_sdi12Mode()
    [...]
}
```

Voyons maintenant en détail ce que font ces quelques lignes.

yocto_api.php et yocto_sdi12port.php

Ces deux includes PHP permettent d'avoir accès aux fonctions permettant de gérer les modules Yoctopuce. `yocto_api.php` doit toujours être inclus, `yocto_sdi12port.php` est nécessaire pour gérer les modules contenant un port SDI12, comme le Yocto-SDI12.

YAPI::RegisterHub

La fonction `YAPI::RegisterHub` permet d'indiquer sur quelle machine se trouve les modules Yoctopuce, ou plus exactemenent sur quelle machine tourne le programme VirtualHub. Dans notre cas l'adresse `127.0.0.1:4444` indique la machine locale, en utilisant le port `4444` (le port standard utilisé par Yoctopuce). Vous pouvez parfaitement changer cette adresse, et mettre l'adresse d'une autre machine sur laquelle tournerait un autre VirtualHub.

YSdi12Port::FindSdi12Port

La fonction `YSdi12Port::FindSdi12Port` permet de retrouver un port SDI12 en fonction du numéro de série de son module hôte et de son nom de fonction. Mais vous pouvez tout aussi bien utiliser des noms logiques que vous auriez préalablement configurés. Imaginons un module Yocto-SDI12 avec le numéros de série *YSDIMK01-123456* que vous auriez appelé "*MonModule*" et dont vous auriez nommé la fonction *sdi12Port* "*MaFonction*", les cinq appels suivants seront strictement équivalents (pour autant que *MaFonction* ne soit définie qu'une fois, pour éviter toute ambiguïté):

```
$sdi12port = YSdi12Port::FindSdi12Port("YSDIMK01-123456.sdi12Port");
$sdi12port = YSdi12Port::FindSdi12Port("YSDIMK01-123456.MaFonction");
$sdi12port = YSdi12Port::FindSdi12Port("MonModule.sdi12Port");
$sdi12port = YSdi12Port::FindSdi12Port("MonModule.MaFonction");
$sdi12port = YSdi12Port::FindSdi12Port("MaFonction");
```

YSdi12Port::FindSdi12Port renvoie un objet que vous pouvez ensuite utiliser à loisir pour contrôler le port SDI12.

isOnline

La méthode isOnline() de l'objet renvoyé par YSdi12Port::FindSdi12Port permet de savoir si le module correspondant est présent et en état de marche.

reset

La méthode reset() de l'objet returné par yFindSerialPort vide tous les tampons du port série.

discoverSingleSensor

La méthode discoverSingleSensor() cherche l'adresse du capteur connecté sur le port SDI-12 et renvoie un objet avec toute les informations du capteur.

readSensor

La méthode readSensor() transmet la commande spécifiée sur le port SDI-12 au capteur spécifié et renvoie une liste d'objet avec toute les valeurs envoyées par le capteur.

Un exemple réel

Ouvrez votre éditeur de texte préféré[4], recopiez le code ci dessous, sauvez-le dans un répertoire accessible par votre serveur web/PHP avec les fichiers de la librairie, et ouvrez-la page avec votre browser favori. Vous trouverez aussi ce code dans le répertoire **Examples/Doc-GettingStarted-Yocto-SDI12** de la librairie Yoctopuce.

Vous reconnaîtrez dans cet exemple l'utilisation des fonctions expliquées ci-dessus, cette fois utilisées avec le décorum nécessaire à en faire un petit programme d'exemple concret.

```
<HTML>
<HEAD>
 <TITLE>Hello World</TITLE>
</HEAD>
<BODY>
<?php
  include('../../php8/yocto_api.php');
  include('../../php8/yocto_sdi12port.php');

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

  // Setup the API to use the VirtualHub on local machine
  $errmsg = '';
  if(YAPI::RegisterHub('127.0.0.1',$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
      $sdi12port = YSdi12Port::FindSdi12Port("$serial.sdi12Port");
      if (!$sdi12port->isOnline()) {
          die("Module not connected (check serial and USB cable)");
      }
  } else {
      // or use any connected module suitable for the demo
      $sdi12port = YSdi12Port::FirstSdi12Port();
      if(is_null($sdi12port)) {
          die("No module connected (check USB cable)");
      } else {
          $serial = $sdi12port->module()->get_serialnumber();
      }
  }
  Print("Module to use: <input name='serial' value='$serial'><br>\n");
```

[4] Si vous n'avez pas d'éditeur de texte, utilisez Notepad plutôt que Microsoft Word.

```
  $singleSensor = $sdi12port->discoverSingleSensor();
  Printf("Sensor address : %s <br>\n", $singleSensor->get_sensorAddress());
  Printf("Sensor SDI-12 compatibility : %s <br>\n", $singleSensor->get_sensorProtocol());
  Printf("Sensor company name : %s <br>\n", $singleSensor->get_sensorVendor());
  Printf("Sensor model number : %s <br>\n", $singleSensor->get_sensorModel());
  Printf("Sensor version : %s <br>\n", $singleSensor->get_sensorVersion());
  Printf("Sensor serial number : %s <br>\n", $singleSensor->get_sensorSerial());
  $valSensor = $sdi12port->readSensor($singleSensor->get_sensorAddress(),"M",5000);

  for ($i = 0; $i < sizeof($valSensor); $i++) {
    if ($singleSensor->get_measureCount() > 1) {
       Printf("%s %-8.2f %s %s <br>\n", $singleSensor->get_measureSymbol($i), $valSensor[$i
],
        $singleSensor->get_measureUnit($i), $singleSensor->get_measureDescription($i));
    }
    else{  Printf("%.2f <br>\n", $valSensor[$i]);}

  }
  YAPI::FreeAPI();

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

17.3. Contrôle de la partie module

Chaque module peut-être contrôlé d'une manière similaire, vous trouverez ci dessous un simple programme d'exemple affichant les principaux paramètres d'un module et permettant d'activer la balise de localisation.

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

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

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

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

```
  Print("Module to use: <input name='serial' value='$serial'><br>");

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

Chaque propriété `xxx` du module peut être lue grâce à une méthode du type `get_xxxx()`, et les propriétés qui se sont pas en lecture seule peuvent être modifiées à l'aide de la méthode `set_xxx()` Pour plus de détails concernant ces fonctions utilisées, reportez-vous aux chapitre API

Modifications des réglages du module

Lorsque que vous souhaitez modifier les réglages d'un module, il suffit d'appeler la fonction `set_xxx()` correspondante, cependant cette modification n'a lieu que dans la mémoire vive du module: si le module redémarre, les modifications seront perdues. Pour qu'elle soient mémorisées de manière persistante, il est nécessaire de demander au module de sauvegarder sa configuration courante dans sa mémoire non volatile. Pour cela il faut utiliser la méthode `saveToFlash()`. Inversement il est possible de forcer le module à oublier ses réglages courants en utilisant la méthode `revertFromFlash()`. Ce petit exemple ci-dessous vous permet changer le nom logique d'un 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>
```

Attention, le nombre de cycle d'écriture de la mémoire non volatile du module est limité. Passé cette limite plus rien ne garantit de que la sauvegarde des réglages se passera correctement. Cette limite, lié à la technologie employé par le micro-processeur du module se situe aux alentour de 100000 cycles. Pour résumer vous ne pouvez employer la fonction `saveToFlash()` que 100000 fois au cours de la vie du module. Veillez donc à ne pas appeler cette fonction depuis l'intérieur d'une boucle.

Enumération des modules

Obtenir la liste des modules connectés se fait à l'aide de la fonction `yFirstModule()` qui renvoie le premier module trouvé, il suffit ensuite d'appeler la fonction `nextModule()` de cet objet pour trouver les modules suivants, et ce tant que la réponse n'est pas un `NULL`. Ci-dessous un petit exemple listant les module connectés

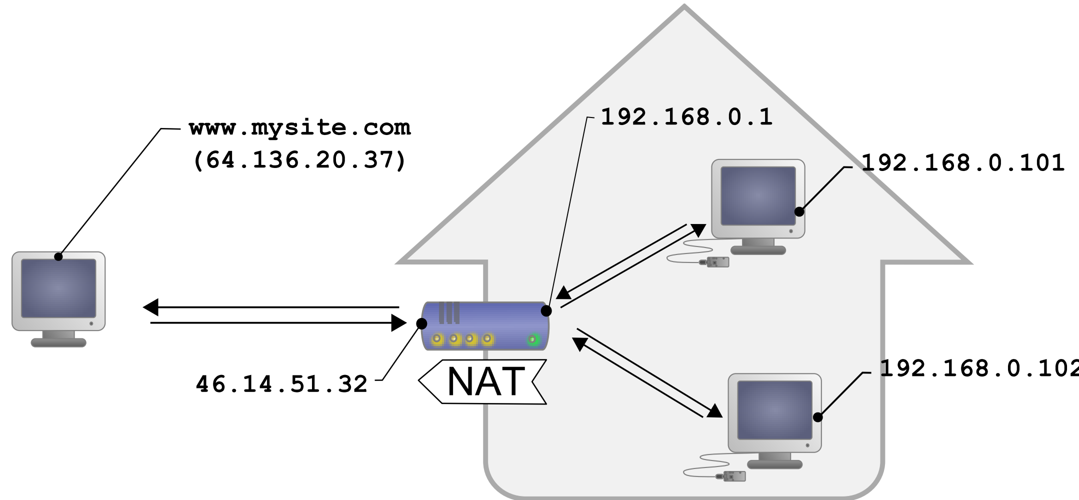
```
<HTML>
<HEAD>
    <TITLE>inventory</TITLE>
</HEAD>
<BODY>
<H1>Device list</H1>
<TT>
    <?php
    include('../../php8/yocto_api.php');
    YAPI::RegisterHub("http://127.0.0.1:4444/");
    $module = YModule::FirstModule();
    while (!is_null($module)) {
        printf("%s (%s)<br>\n", $module->get_serialNumber(),
               $module->get_productName());
        $module=$module->nextModule();
    }
    YAPI::FreeAPI();
    ?>
</TT>
</BODY>
</HTML>
```

17.4. API par callback HTTP et filtres NAT

La librairie PHP est capable de fonctionner dans un mode spécial appelé *Yocto-API par callback HTTP*. Ce mode permet de contrôler des modules Yoctopuce installés derrière un filtre NAT tel qu'un routeur DSL par exemple, et ce sans avoir à un ouvrir un port. L'application typique est le contrôle de modules Yoctopuce situés sur réseau privé depuis un site Web publique.

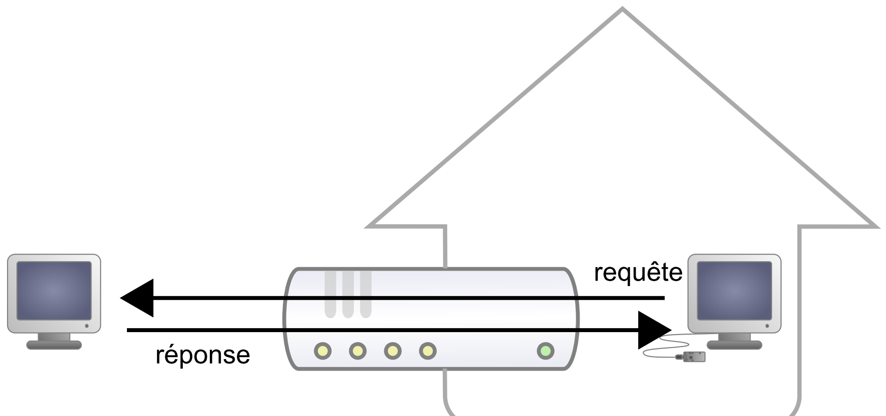
Le filtre NAT, avantages et inconvénients

Un routeur DSL qui effectue de la traduction d'adresse réseau (NAT) fonctionne un peu comme un petit central téléphonique privé: les postes internes peuvent s'appeler l'un l'autre ainsi que faire des appels vers l'extérieur, mais vu de l'extérieur, il n'existe qu'un numéro de téléphone officiel, attribué au central téléphonique lui-même. Les postes internes ne sont pas atteignables depuis l'extérieur.

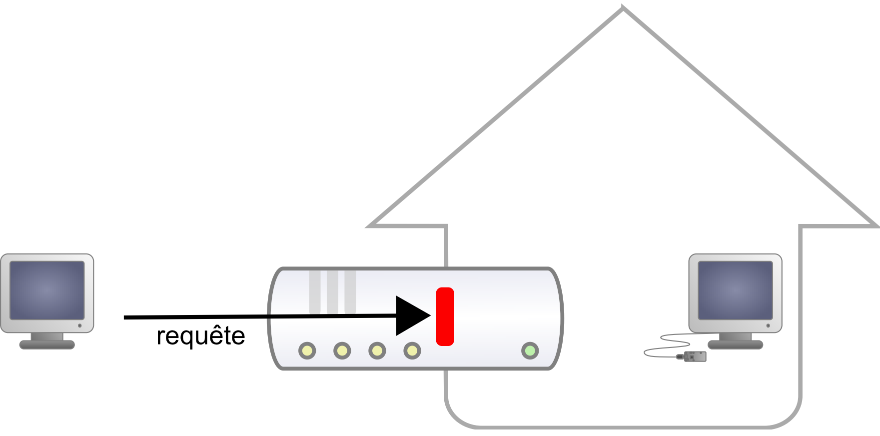


Configuration DSL typique, les machines du LAN sont isolées de l'extérieur par le router DSL

Ce qui, transposé en terme de réseau, donne : les appareils connectés sur un réseau domestique peuvent communiquer entre eux en utilisant une adresse IP locale (du genre 192.168.xxx.yyy), et contacter des serveurs sur Internet par leur adresse publique, mais vu de l'extérieur, il n'y a qu'une seule adresse IP officielle, attribuée au routeur DSL exclusivement. Les différents appareils réseau ne sont pas directement atteignables depuis l'extérieur. C'est assez contraignant, mais c'est une protection relativement efficace contre les intrusions.



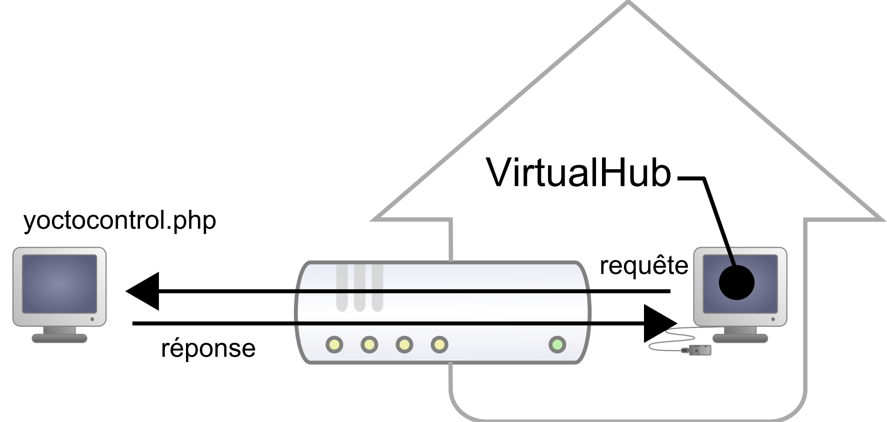
Les réponses aux requêtes venant des machines du LAN sont routées.



Mais les requêtes venant de l'extérieur sont bloquées.

Voir Internet sans être vu représente un avantage de sécurité énorme. Cependant, cela signifie qu'a priori, on ne peut pas simplement monter son propre serveur Web publique chez soi pour une installation domotique et offrir un accès depuis l'extérieur. Une solution à ce problème, préconisée par de nombreux vendeurs de domotique, consiste à donner une visibilité externe au serveur de domotique lui-même, en ouvrant un port et en ajoutant une règle de routage dans la configuration NAT du routeur DSL. Le problème de cette solution est qu'il expose le serveur de domotique aux attaques externes.

L'API par callback HTTP résoud ce problème sans qu'il soit nécessaire de modifier la configuration du routeur DSL. Le script de contrôle des modules est placé sur un site externe, et c'est le *Virtual Hub* qui est chargé de l'appeler à intervalle régulier.



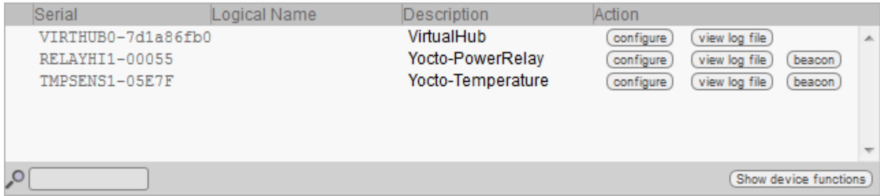
L'API par callback HTTP utilise le VirtualHub, et c'est lui qui initie les requêtes.

Configuration

L'API callback se sert donc du *Virtual Hub* comme passerelle. Toutes les communications sont initiées par le *Virtual Hub*, ce sont donc des communication sortantes, et par conséquent parfaitement autorisée par le routeur DSL.

Il faut configurer le *VirtualHub* pour qu'il appelle le script PHP régulièrement. Pour cela il faut:

1. Lancer un *VirtualHub*
2. Accéder à son interface, généralement 127.0.0.1:4444
3. Cliquer sur le bouton **configure** de la ligne correspondant au *VirtualHub* lui-même
4. Cliquer sur le bouton **edit** de la section **Outgoing callbacks**



Cliquer sur le bouton "configure" de la première ligne

Cliquer sur le bouton "edit" de la section Outgoing callbacks.

Et choisir "Yocto-API callback".

Il suffit alors de définir l'URL du script PHP et, si nécessaire, le nom d'utilisateur et le mot de passe pour accéder à cette URL. Les méthodes d'authentification supportées sont *basic* et *digest*. La seconde est plus sûre que la première car elle permet de ne pas transférer le mot de passe sur le réseau.

Utilisation

Du point de vue du programmeur, la seule différence se trouve au niveau de l'appel à la fonction *yRegisterHub*; au lieu d'utiliser une adresse IP, il faut utiliser la chaîne *callback* (ou *http://callback*, qui est équivalent).

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

La suite du code reste strictement identique. Sur l'interface du *VirtualHub*, il y a en bas de la fenêtre de configuration de l'API par callback HTTP un bouton qui permet de tester l'appel au script PHP.

Il est à noter que le script PHP qui contrôle les modules à distance via l'API par callback HTTP ne peut être appelé que par le *VirtualHub*. En effet, il a besoin des informations postées par le *VirtualHub* pour fonctionner. Pour coder un site Web qui contrôle des modules Yoctopuce de manière interactive, il faudra créer une interface utilisateur qui stockera dans un fichier ou une base de données les actions à effectuer sur les modules Yoctopuce. Ces actions seront ensuite lues puis exécutés par le script de contrôle.

Problèmes courants

Pour que l'API par callback HTTP fonctionne, l'option de PHP *allow_url_fopen* doit être activée. Certains hébergeurs de site web ne l'activent pas par défaut. Le problème se manifeste alors avec l'erreur suivante:

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

Pour activer cette option, il suffit de créer dans le même répertoire que le script PHP de contrôle un fichier *.htaccess* contenant la ligne suivante:

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

Selon la politique de sécurité de l'hébergeur, il n'est parfois pas possible d'autoriser cette option à la racine du site web, où même d'installer des scripts PHP recevant des données par un POST HTTP. Dans ce cas il suffit de placer le script PHP dans un sous-répertoire.

Limitations

Cette méthode de fonctionnement qui permet de passer les filtres NAT à moindre frais a malgré tout un prix. Les communications étant initiées par le *Virtual Hub* à intervalle plus ou moins régulier, le temps de réaction à un événement est nettement plus grand que si les modules Yoctopuce étaient pilotés en direct. Vous pouvez configurer le temps de réaction dans la fenêtre ad-hoc du *Virtual Hub*, mais il sera nécessairement de quelques secondes dans le meilleur des cas.

Le mode *Yocto-API par callback HTTP* n'est pour l'instant disponible qu'en PHP, EcmaScript (Node.JS) et Java.

17.5. Gestion des erreurs

Lorsque vous implémentez un programme qui doit interagir avec des modules USB, vous ne pouvez pas faire abstraction de la gestion des erreurs. Il y aura forcément une occasion où un utilisateur aura débranché le périphérique, soit avant de lancer le programme, soit même en pleine opération. La librairie Yoctopuce est prévue pour vous aider à supporter ce genre de comportements, mais votre code doit néanmoins être fait pour se comporter au mieux pour interpréter les erreurs signalées par la librairie.

La manière la plus simple de contourner le problème est celle que nous avons employé pour les petits exemples précédents de ce chapitre: avant d'accéder à un module, on vérifie qu'il est en ligne avec la méthode `isOnline()` et on suppose ensuite qu'il va y rester pendant la fraction de seconde nécessaire à exécuter les lignes de code suivantes. Ce n'est pas parfait, mais ça peut suffire dans certains cas. Il faut toutefois être conscient qu'on ne peut pas totalement exclure une erreur se produisant après le `isOnline()`, qui pourrait faire planter le programme. La seule manière de l'éviter est d'implémenter une des deux techniques de gestion des erreurs décrites ci-dessous.

La méthode recommandée par la plupart des langages de programmation pour la gestion des erreurs imprévisibles est l'utilisation d'exceptions. C'est le comportement par défaut de la librairie Yoctopuce. Si une erreur se produit alors qu'on essaie d'accéder à un module, la librairie va lancer une exception. Dans ce cas, de trois choses l'une:

- Si votre code attrape l'exception au vol et la gère, et tout se passe bien.
- Si votre programme tourne dans le debugger, vous pourrez relativement facilement déterminer où le problème s'est produit, et voir le message explicatif lié à l'exception.
- Sinon... l'exception va crasher votre programme, boum!

Comme cette dernière situation n'est pas la plus souhaitable, la librairie Yoctopuce offre une autre alternative pour la gestion des erreurs, permettant de faire un programme robuste sans devoir attraper les exceptions à chaque ligne de code. Il suffit d'appeler la fonction `YAPI.DisableExceptions()` pour commuter la librairie dans un mode où les exceptions de chaque fonction sont systématiquement remplacées par des valeurs de retour particulières, qui

peuvent être testées par l'appelant lorsque c'est pertinent. Le nom de la valeur de retour en cas d'erreur pour chaque fonction est systématiquement documenté dans la référence de la librairie. Il suit toujours la même logique: une méthode `get_state()` retournera une valeur `NomDeClasse.STATE_INVALID`, une méthode `get_currentValue` retournera une valeur `NomDeClasse.CURRENTVALUE_INVALID`, etc. Dans tous les cas, la valeur retournée sera du type attendu, et ne sera pas un pointeur nul qui risquerait de faire crasher votre programme. Au pire, si vous affichez la valeur sans la tester, elle sera hors du cadre attendu pour la valeur retournée. Dans le cas de fonctions qui ne retournent à priori pas d'information, la valeur de retour sera `YAPI.SUCCESS` si tout va bien, et un code d'erreur différent en cas d'échec.

Quand vous travaillez sans les exceptions, il est possible d'obtenir un code d'erreur et un message expliquant l'origine de l'erreur en le demandant à l'objet qui a retourné une erreur à l'aide des méthodes `errType()` et `errMessage()`. Ce sont les même informations qui auraient été associées à l'exception si elles avaient été actives.

18. Utilisation du Yocto-SDI12 en VisualBasic .NET

VisualBasic a longtemps été la porte d'entrée privilégiée vers le monde Microsoft. Nous nous devions donc d'offrir notre interface pour ce langage, même si la nouvelle tendance est le C#. Nous supportons Visual Studio 2017 et les versions plus récentes.

18.1. Installation

Téléchargez la librairie Yoctopuce pour Visual Basic depuis le site web de Yoctopuce[1]. Il n'y a pas de programme d'installation, copiez simplement de contenu du fichier zip dans le répertoire de votre choix. Vous avez besoin essentiellement du contenu du répertoire `Sources`. Les autres répertoires contiennent la documentation et quelques programmes d'exemple. Les projets d'exemple sont des projets Visual Basic 2010, si vous utilisez une version antérieure, il est possible que vous ayez à reconstruire la structure de ces projets.

18.2. Utilisation l'API yoctopuce dans un projet Visual Basic

La librairie Yoctopuce pour Visual Basic .NET se présente sous la forme d'une DLL et de fichiers sources en Visual Basic. La DLL n'est pas une DLL .NET mais une DLL classique, écrite en C, qui gère les communications à bas niveau avec les modules[2]. Les fichiers sources en Visual Basic gèrent la partie haut niveau de l'API. Vous avez donc besoin de cette DLL et des fichiers .vb du répertoire `Sources` pour créer un projet gérant des modules Yoctopuce.

Configuration d'un projet Visual Basic

Les indications ci-dessous sont fournies pour Visual Studio express 2010, mais la procédure est semblable pour les autres versions.

Commencez par créer votre projet, puis depuis le panneau **Explorateur de solutions** effectuez un clic droit sur votre projet, et choisissez **Ajouter** puis **Elément existant**.

Une fenêtre de sélection de fichiers apparaît: sélectionnez le fichier `yocto_api.vb` et les fichiers correspondant aux fonctions des modules Yoctopuce que votre projet va gérer. Dans le doute, vous pouvez aussi sélectionner tous les fichiers.

Vous avez alors le choix entre simplement ajouter ces fichiers à votre projet, ou les ajouter en tant que lien (le bouton **Ajouter** est en fait un menu déroulant). Dans le premier cas, Visual Studio va copier les fichiers choisis dans votre projet, dans le second Visual Studio va simplement garder un

[1] www.yoctopuce.com/FR/libraries.php
[2] Les sources de cette DLL sont disponibles dans l'API C++

lien sur les fichiers originaux. Il est recommandé d'utiliser des liens, une éventuelle mise à jour de la librairie sera ainsi beaucoup plus facile.

Ensuite, ajoutez de la même manière la dll `yapi.dll`, qui se trouve dans le répertoire `Sources/dll`[3]. Puis depuis la fenêtre **Explorateur de solutions**, effectuez un clic droit sur la DLL, choisissez **Propriété** et dans le panneau **Propriétés**, mettez l'option **Copier dans le répertoire de sortie** à **toujours copier**. Vous êtes maintenant prêt à utiliser vos modules Yoctopuce depuis votre environnement Visual Studio.

Afin de les garder simples, tous les exemples fournis dans cette documentation sont des applications consoles. Il va de soit que que les fonctionnement des librairies est strictement identiques si vous les intégrez dans une application dotée d'une interface graphique.

18.3. Contrôle de la fonction Sdi12Port

Il suffit de quelques lignes de code pour piloter un Yocto-SDI12. Voici le squelette d'un fragment de code VisualBasic .NET qui utilise la fonction Sdi12Port.

```
[...]
' On active la détection des modules sur USB
Dim errmsg As String
YAPI.RegisterHub("usb", errmsg)
[...]

' On récupère l'objet permettant d'intéragir avec le module
Dim sdi12port As YSdi12Port
sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.sdi12Port")

' Pour gérer le hot-plug, on vérifie que le module est là
If (sdi12port.isOnline()) Then
    ' Utiliser sdi12port.set_sdi12Mode()
    [...]
End If

[...]
```

Voyons maintenant en détail ce que font ces quelques lignes.

YAPI.RegisterHub

La fonction `YAPI.RegisterHub` initialise l'API de Yoctopuce en indiquant où les modules doivent être recherchés. Utilisée avec le paramètre `"usb"`, elle permet de travailler avec les modules connectés localement à la machine. Si l'initialisation se passe mal, cette fonction renverra une valeur différente de `YAPI_SUCCESS`, et retournera via le paramètre `errmsg` un explication du problème.

YSdi12Port.FindSdi12Port

La fonction `YSdi12Port.FindSdi12Port` permet de retrouver un port SDI12 en fonction du numéro de série de son module hôte et de son nom de fonction. Mais vous pouvez tout aussi bien utiliser des noms logiques que vous auriez préalablement configurés. Imaginons un module Yocto-SDI12 avec le numéros de série *YSDIMK01-123456* que vous auriez appelé "*MonModule*" et dont vous auriez nommé la fonction *sdi12Port* "*MaFonction*", les cinq appels suivants seront strictement équivalents (pour autant que *MaFonction* ne soit définie qu'une fois, pour éviter toute ambiguïté):

```
sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.sdi12Port")
sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.MaFonction")
sdi12port = YSdi12Port.FindSdi12Port("MonModule.sdi12Port")
sdi12port = YSdi12Port.FindSdi12Port("MonModule.MaFonction")
sdi12port = YSdi12Port.FindSdi12Port("MaFonction")
```

`YSdi12Port.FindSdi12Port` renvoie un objet que vous pouvez ensuite utiliser à loisir pour contrôler le port SDI12.

[3] Pensez à changer le filtre de la fenêtre de sélection de fichiers, sinon la DLL n'apparaîtra pas

isOnline

La méthode `isOnline()` de l'objet renvoyé par `YSdi12Port.FindSdi12Port` permet de savoir si le module correspondant est présent et en état de marche.

reset

La méthode `reset()` de l'objet returné par `yFindSerialPort` vide tous les tampons du port série.

discoverSingleSensor

La méthode `discoverSingleSensor()` cherche l'adresse du capteur connecté sur le port SDI-12 et renvoie un objet avec toute les informations du capteur.

readSensor

La méthode `readSensor()` transmet la commande spécifiée sur le port SDI-12 au capteur spécifié et renvoie une liste d'objet avec toute les valeurs envoyées par le capteur.

Un exemple réel

Lancez Microsoft VisualBasic et ouvrez le projet exemple correspondant, fourni dans le répertoire **Examples/Doc-GettingStarted-Yocto-SDI12** de la librairie Yoctopuce.

Vous reconnaîtrez dans cet exemple l'utilisation des fonctions expliquées ci-dessus, cette fois utilisées avec le décorum nécessaire à en faire un petit programme d'exemple concret.

```
Module Module1

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

  Sub Main()
    Dim argv() As String = System.Environment.GetCommandLineArgs()
    Dim errmsg As String = ""
    Dim target As String
        Dim sdi12Port As YSdi12Port

        If argv.Length < 1 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
            sdi12Port = YSdi12Port.FirstSdi12Port()
            If sdi12Port Is Nothing Then
                Console.WriteLine("No module connected (check USB cable) ")
                End
            End If
            target = sdi12Port.get_module().get_serialNumber()
        End If
        sdi12Port = YSdi12Port.FindSdi12Port(target + ".sdi12Port")
        While True
            If (sdi12Port.isOnline()) Then
                Console.SetCursorPosition(0, 0)
                Dim singleSensor As YSdi12SensorInfo = sdi12Port.discoverSingleSensor()
                Console.WriteLine("Sensor address : " + singleSensor.get_sensorAddress())
                Console.WriteLine("Sensor SDI-12 compatibility : " +
singleSensor.get_sensorProtocol())
                Console.WriteLine("Sensor company name : " + singleSensor.get_sensorVendor(
```

```
))
                Console.WriteLine("Sensor model number : " + singleSensor.get_sensorModel()
)
                Console.WriteLine("Sensor version : " + singleSensor.get_sensorVersion())
                Console.WriteLine("Sensor serial number : " + singleSensor.get_sensorSerial
())
                Dim valSensor As List(Of Double) = sdi12Port.readSensor
(singleSensor.get_sensorAddress(), "M", 5000)
                Dim i = 0
                While (i < valSensor.Count)
                    If singleSensor.get_measureCount() > 1 Then
                        Console.WriteLine(String.Format("{0} : {1:0.00} {2} {3}",
singleSensor.get_measureSymbol(i), valSensor(i),
                                                singleSensor.get_measureUnit(i),
singleSensor.get_measureDescription(i)))
                    Else
                        Console.WriteLine(valSensor(i))
                    End If
                    i = i + 1
                End While
            Else
                Console.WriteLine("Module not connected (check identification and USB
cable)")
            End If

            REM wait 5 sec to show the output
            System.Threading.Thread.Sleep(5000)
        End While
        YAPI.FreeAPI()
  End Sub

End Module
```

18.4. Contrôle de la partie module

Chaque module peut-être contrôlé d'une manière similaire, vous trouverez ci dessous un simple programme d'exemple affichant les principaux paramètres d'un module et permettant d'activer la balise de localisation.

```
Imports System.IO
Imports System.Environment

Module Module1

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

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

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

    If argv.Length < 2 Then usage()

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

```
      Else
        Console.WriteLine("OFF")
      End If
      Console.WriteLine("upTime:       " + Str(m.get_upTime() / 1000) + " sec")
      Console.WriteLine("USB current:  " + Str(m.get_usbCurrent()) + " mA")
      Console.WriteLine("Logs:")
      Console.WriteLine(m.get_lastLogs())
    Else
      Console.WriteLine(argv(1) + " not connected (check identification and USB cable)")
    End If
    YAPI.FreeAPI()
  End Sub

End Module
```

Chaque propriété `xxx` du module peut être lue grâce à une méthode du type `get_xxxx()`, et les propriétés qui se sont pas en lecture seule peuvent être modifiées à l'aide de la méthode `set_xxx()` Pour plus de détails concernant ces fonctions utilisées, reportez-vous aux chapitre API

Modifications des réglages du module

Lorsque que vous souhaitez modifier les réglages d'un module, il suffit d'appeler la fonction `set_xxx()` correspondante, cependant cette modification n'a lieu que dans la mémoire vive du module: si le module redémarre, les modifications seront perdues. Pour qu'elle soient mémorisées de manière persistante, il est nécessaire de demander au module de sauvegarder sa configuration courante dans sa mémoire non volatile. Pour cela il faut utiliser la méthode `saveToFlash()`. Inversement il est possible de forcer le module à oublier ses réglages courants en utilisant la méthode `revertFromFlash()`. Ce petit exemple ci-dessous vous permet changer le nom logique d'un module.

```
Module Module1


  Sub usage()

    Console.WriteLine("usage: demo <serial or logical name> <new logical name>")
    End
  End Sub

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

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

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

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

  End Sub

End Module
```

Attention, le nombre de cycles d'écriture de la mémoire non volatile du module est limité. Passé cette limite plus rien ne garantit que la sauvegarde des réglages se passera correctement. Cette limite, liée à la technologie employée par le micro-processeur du module se situe aux alentour de 100000 cycles. Pour résumer vous ne pouvez employer la fonction `saveToFlash()` que 100000 fois au cours de la vie du module. Veillez donc à ne pas appeler cette fonction depuis l'intérieur d'une boucle.

Enumeration des modules

Obtenir la liste des modules connectés se fait à l'aide de la fonction `yFirstModule()` qui renvoie le premier module trouvé, il suffit ensuite d'appeler la fonction `nextModule()` de cet objet pour trouver les modules suivants, et ce tant que la réponse n'est pas un `Nothing`. Ci-dessous un petit exemple listant les module connectés

```
Module Module1

  Sub Main()
    Dim M As ymodule
    Dim errmsg As String = ""

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

    Console.WriteLine("Device list")
    M = YModule.FirstModule()
    While M IsNot Nothing
      Console.WriteLine(M.get_serialNumber() + " (" + M.get_productName() + ")")
      M = M.nextModule()
    End While
    YAPI.FreeAPI()
  End Sub

End Module
```

18.5. Gestion des erreurs

Lorsque vous implémentez un programme qui doit interagir avec des modules USB, vous ne pouvez pas faire abstraction de la gestion des erreurs. Il y aura forcément une occasion où un utilisateur aura débranché le périphérique, soit avant de lancer le programme, soit même en pleine opération. La librairie Yoctopuce est prévue pour vous aider à supporter ce genre de comportements, mais votre code doit néanmoins être fait pour se comporter au mieux pour interpréter les erreurs signalées par la librairie.

La manière la plus simple de contourner le problème est celle que nous avons employé pour les petits exemples précédents de ce chapitre: avant d'accéder à un module, on vérifie qu'il est en ligne avec la méthode `isOnline()` et on suppose ensuite qu'il va y rester pendant la fraction de seconde nécessaire à exécuter les lignes de code suivantes. Ce n'est pas parfait, mais ça peut suffire dans certains cas. Il faut toutefois être conscient qu'on ne peut pas totalement exclure une erreur se produisant après le `isOnline()`, qui pourrait faire planter le programme. La seule manière de l'éviter est d'implémenter une des deux techniques de gestion des erreurs décrites ci-dessous.

La méthode recommandée par la plupart des langages de programmation pour la gestion des erreurs imprévisibles est l'utilisation d'exceptions. C'est le comportement par défaut de la librairie Yoctopuce. Si une erreur se produit alors qu'on essaie d'accéder à un module, la librairie va lancer une exception. Dans ce cas, de trois choses l'une:

- Si votre code attrape l'exception au vol et la gère, et tout se passe bien.
- Si votre programme tourne dans le debugger, vous pourrez relativement facilement déterminer où le problème s'est produit, et voir le message explicatif lié à l'exception.

- Sinon... l'exception va crasher votre programme, boum!

Comme cette dernière situation n'est pas la plus souhaitable, la librairie Yoctopuce offre une autre alternative pour la gestion des erreurs, permettant de faire un programme robuste sans devoir attraper les exceptions à chaque ligne de code. Il suffit d'appeler la fonction `YAPI.DisableExceptions()` pour commuter la librairie dans un mode où les exceptions de chaque fonction sont systématiquement remplacées par des valeurs de retour particulières, qui peuvent être testées par l'appelant lorsque c'est pertinent. Le nom de la valeur de retour en cas d'erreur pour chaque fonction est systématiquement documenté dans la référence de la librairie. Il suit toujours la même logique: une méthode `get_state()` retournera une valeur `NomDeClasse.STATE_INVALID`, une méthode `get_currentValue` retournera une valeur `NomDeClasse.CURRENTVALUE_INVALID`, etc. Dans tous les cas, la valeur retournée sera du type attendu, et ne sera pas un pointeur nul qui risquerait de faire crasher votre programme. Au pire, si vous affichez la valeur sans la tester, elle sera hors du cadre attendu pour la valeur retournée. Dans le cas de fonctions qui ne retournent à priori pas d'information, la valeur de retour sera `YAPI.SUCCESS` si tout va bien, et un code d'erreur différent en cas d'échec.

Quand vous travaillez sans les exceptions, il est possible d'obtenir un code d'erreur et un message expliquant l'origine de l'erreur en le demandant à l'objet qui a retourné une erreur à l'aide des méthodes `errType()` et `errMessage()`. Ce sont les même informations qui auraient été associées à l'exception si elles avaient été actives.

19. Utilisation du Yocto-SDI12 en Delphi / Lazarus

Delphi est l'héritier de Turbo-Pascal. A l'origine, Delphi était produit par Borland, mais c'est maintenant Embarcadero qui l'édite. Sa force réside dans sa facilité d'utilisation, il permet à quiconque ayant des notions de Pascal de programmer une application Windows en deux temps trois mouvements. Son seul défaut est d'être payant[1].

Lazarus[2] est un IDE gratuit basé sur Free-Pascal qui n'a pas grand chose à envier à Delphi. Il a aussi l'avantage d'exister pour Windows et Linux. La librairie Yoctopuce pour Delphi est compatible avec Lazarus tant sous Windows que Linux.

Les librairies pour Delphi / Lazarus sont fournies non pas sous forme de composants VCL, mais directement sous forme de fichiers source. Ces fichiers sont compatibles avec la plupart des versions de Delphi / Lazarus [3].

19.1. Préparation

Connectez-vous sur le site de Yoctopuce et téléchargez la la librairie Yoctopuce pour Delphi[4]. Décompressez le tout dans le répertoire de votre choix.

- Avec Delphi ajoutez le sous-répertoire *sources* de l'archive dans la liste des répertoires des librairies de Delphi[5].
- Avec Lazarus, ouvrez les options de votre projet et ajoutez le répertoire *sources* dans le champs "other unit files"[6].

Windows

Sous Windows, la librairie Delphi / Lazarus utilise deux DLL: *yapi.dll* pour exécutables 32bits et *yapi64.dll* pour les exécutable 64bits. Toutes les applications que vous créerez avec Delphi ou Lazarus devront avoir accès à ces DLL. Le plus simple est de faire en sorte qu'elles soient présentes dans le même répertoire que l'exécutable de votre application. Vous trouverez ces DLL dans le répertoire *sources/dll*.

---

[1] En fait, Borland a diffusé des versions gratuites (pour usage personnel) de Delphi 2006 et Delphi 2007, en cherchant un peu sur internet il est encore possible de les télécharger.
[2] www.lazarus-ide.org
[3] Les librairies Delphi sont régulièrement testées avec Delphi 5 et Delphi XE2 et la dernière version de Lazarus
[4] www.yoctopuce.com/FR/libraries.php
[5] Utilisez le menu **outils / options denvironnement**
[6] Utilisez le menu **Project / Project options/ Compiler options / Paths**

Linux

Sous Linux, la librairie Delphi / Lazarus utilise les librairies suivantes:

- libyapi-i386.so sur les systèmes Intel 32 bits
- libyapi-amd64.so sur les systèmes Intel 64 bits
- libyapi-armhf.so sur les systèmes ARM 32 bits
- libyapi-aarch64.so sur les systèmes ARM 64 bits

Vous trouverez ces fichiers lib dans le répertoire *sources/dll*. Vous devez faire en sorte que :

- Lazarus soit capable de localiser le bon fichier .so à la compilation
- Lexécutable soit capable de le localiser l'exécution

La solution la plus simple pour remplir ces conditions consiste à copier ces quatre fichiers dans le répertoire */usr/lib*. Une autre solution consiste à les copier dans le même répertoire que votre code source et à ajuster votre variable d'environnement *LD_LIBRARY_PATH*en conséquence.

A propos des exemples

Afin de les garder simples, tous les exemples fournis dans cette documentation sont des applications consoles. Il va de soit que le fonctionnement des librairies est strictement identique avec des applications fenêtrées.

Notez que la plupart de ces exemples utilisent des paramètres passés sur le ligne de commande[7].

Vous allez rapidement vous rendre compte que l'API Delphi définit beaucoup de fonctions qui retournent des objets. Vous ne devez jamais désallouer ces objets vous-même. Ils seront désalloués automatiquement par l'API à la fin de l'application.

19.2. Contrôle de la fonction Sdi12Port

Il suffit de quelques lignes de code pour piloter un Yocto-SDI12. Voici le squelette d'un fragment de code Delphi qui utilise la fonction Sdi12Port.

```
uses yocto_api, yocto_sdi12port;

var errmsg: string;
    sdi12port: TYSdi12Port;

[...]
// On active la détection des modules sur USB
yRegisterHub('usb',errmsg)
[...]

// On récupère l'objet permettant d'intéragir avec le module
sdi12port = yFindSdi12Port("YSDIMK01-123456.sdi12Port")

// Pour gérer le hot-plug, on vérifie que le module est là
if sdi12port.isOnline() then
    begin
        // use sdi12port.set_sdi12Mode()
        [...]
    end;
[...]
```

Voyons maintenant en détail ce que font ces quelques lignes.

yocto_api et yocto_sdi12port

Ces deux unités permettent d'avoir accès aux fonctions permettant de gérer les modules Yoctopuce. `yocto_api` doit toujours être utilisé, `yocto_sdi12port` est nécessaire pour gérer les modules contenant un port SDI12, comme le Yocto-SDI12.

[7] voir http://www.yoctopuce.com/FR/article/a-propos-des-programmes-d-exemples

yRegisterHub

La fonction `yRegisterHub` initialise l'API de Yoctopuce en indiquant où les modules doivent être recherchés. Utilisée avec le paramètre `'usb'`, elle permet de travailler avec les modules connectés localement à la machine. Si l'initialisation se passe mal, cette fonction renverra une valeur différente de `YAPI_SUCCESS`, et retournera via le paramètre `errmsg` un explication du problème.

yFindSdi12Port

La fonction `yFindSdi12Port` permet de retrouver un port SDI12 en fonction du numéro de série de son module hôte et de son nom de fonction. Mais vous pouvez tout aussi bien utiliser des noms logiques que vous auriez préalablement configurés. Imaginons un module Yocto-SDI12 avec le numéros de série *YSDIMK01-123456* que vous auriez appelé "*MonModule*" et dont vous auriez nommé la fonction *sdi12Port* "*MaFonction*", les cinq appels suivants seront strictement équivalents (pour autant que *MaFonction* ne soit définie qu'une fois, pour éviter toute ambiguïté):

```
sdi12port := yFindSdi12Port("YSDIMK01-123456.sdi12Port");
sdi12port := yFindSdi12Port("YSDIMK01-123456.MaFonction");
sdi12port := yFindSdi12Port("MonModule.sdi12Port");
sdi12port := yFindSdi12Port("MonModule.MaFonction");
sdi12port := yFindSdi12Port("MaFonction");
```

`yFindSdi12Port` renvoie un objet que vous pouvez ensuite utiliser à loisir pour contrôler le port SDI12.

isOnline

La méthode `isOnline()` de l'objet renvoyé par `yFindSdi12Port` permet de savoir si le module correspondant est présent et en état de marche.

reset

La méthode `reset()` de l'objet returné par `yFindSerialPort` vide tous les tampons du port série.

discoverSingleSensor

La méthode `discoverSingleSensor()` cherche l'adresse du capteur connecté sur le port SDI-12 et renvoie un objet avec toute les informations du capteur.

readSensor

La méthode `readSensor()` transmet la commande spécifiée sur le port SDI-12 au capteur spécifié et renvoie une liste d'objet avec toute les valeurs envoyées par le capteur.

Un exemple réel

Lancez votre environnement Delphi, copiez la DLL yapi.dll dans un répertoire et créez une nouvelle application console dans ce même répertoire, et copiez-coller le code ci dessous.

Vous reconnaîtrez dans cet exemple l'utilisation des fonctions expliquées ci-dessus, cette fois utilisées avec le décorum nécessaire à en faire un petit programme d'exemple concret.

```
program demo;
{$APPTYPE CONSOLE}
uses
  SysUtils,
  {$IFNDEF UNIX}
  windows,
  {$ENDIF UNIX}

  yocto_api,
  yocto_sdi12port;

procedure usage();
  var
    execname:string;
  begin
```

```
    execname := ExtractFileName(paramstr(0));
    WriteLn('Usage:');
    WriteLn(execname + ' <serial_number>');
    WriteLn(execname + ' <logical_name> ');
    WriteLn(execname + ' any           (use any discovered device)');
    sleep(3000);
    halt;
  end;

var
 errmsg,target : string;
 m : TYModule;
 sdi12Port : TYSdi12Port;
 singleSensor : TYSdi12SensorInfo;
 valSensor : TDoubleArray;
 j :integer;

begin
  if (paramcount<1) then usage();
  target := UpperCase(paramstr(1));

  if (YRegisterHub('usb', errmsg) <> YAPI_SUCCESS)  then
    begin
      writeln('RegisterHub error: ' + errmsg);
      halt;
    end;

  if (target='ANY') then
    begin
      sdi12Port := YFirstSdi12Port();
      if (sdi12Port = nil) then
       begin
         writeln('No module connected (check USB cable)');
         sleep(3000);
         halt;
       end;
      m := sdi12Port.get_module();
      target := m.get_serialNumber();
     end;

  Writeln(target);
  sdi12Port := YFindSdi12Port(target + '.sdi12Port');

  if (sdi12Port.isOnline()) then
    begin
        singleSensor := sdi12Port.discoverSingleSensor();
        writeln('Sensor address : ' + singleSensor.get_sensorAddress());
        writeln('Sensor SDI-12 compatibility : ' + singleSensor.get_sensorProtocol());
        writeln('Sensor company name : ' +  singleSensor.get_sensorVendor());
        writeln('Sensor model number : ' + singleSensor.get_sensorModel());
        writeln('Sensor version : ' + singleSensor.get_sensorVersion());
        writeln('Sensor serial number : ' + singleSensor.get_sensorSerial());
        valSensor := sdi12Port.readSensor(singleSensor.get_sensorAddress(),'M', 5000);

        for j := 0 to length(valSensor)-1 do
        begin
                writeln(Format('%s: %.2f %s %s',
            [singleSensor.get_measureSymbol(j), valSensor[j],
            singleSensor.get_measureUnit(j), singleSensor.get_measureDescription(j)]));
        end;
        sleep(5000);


    end
  else writeln('Module not connected (check identification and USB cable)');

  yFreeAPI();
end.
```

19.3. Contrôle de la partie module

Chaque module peut-être contrôlé d'une manière similaire, vous trouverez ci dessous un simple programme d'exemple affichant les principaux paramètres d'un module et permettant d'activer la balise de localisation.

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

const
  serial = 'YSDIMK01-123456'; // 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.
```

Chaque propriété `xxx` du module peut être lue grâce à une méthode du type `get_xxxx()`, et les propriétés qui se sont pas en lecture seule peuvent être modifiées à l'aide de la méthode `set_xxx()` Pour plus de détails concernant ces fonctions utilisées, reportez-vous aux chapitre API

Modifications des réglages du module

Lorsque que vous souhaitez modifier les réglages d'un module, il suffit d'appeler la fonction `set_xxx()` correspondante, cependant cette modification n'a lieu que dans la mémoire vive du module: si le module redémarre, les modifications seront perdues. Pour qu'elle soient mémorisées de manière persistante, il est nécessaire de demander au module de sauvegarder sa configuration courante dans sa mémoire non volatile. Pour cela il faut utiliser la méthode `saveToFlash()`.

Inversement il est possible de forcer le module à oublier ses réglages courants en utilisant la méthode `revertFromFlash()`. Ce petit exemple ci-dessous vous permet changer le nom logique d'un module.

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

const
  serial = 'YSDIMK01-123456'; // 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.
```

Attention, le nombre de cycles d'écriture de la mémoire non volatile du module est limité. Passé cette limite plus rien ne garantit que la sauvegarde des réglages se passera correctement. Cette limite, liée à la technologie employée par le micro-processeur du module se situe aux alentour de 100000 cycles. Pour résumer vous ne pouvez employer la fonction `saveToFlash()` que 100000 fois au cours de la vie du module. Veillez donc à ne pas appeler cette fonction depuis l'intérieur d'une boucle.

Énumération des modules

Obtenir la liste des modules connectés se fait à l'aide de la fonction `yFirstModule()` qui renvoie le premier module trouvé, il suffit ensuite d'appeler la fonction `nextModule()` de cet objet pour trouver les modules suivants, et ce tant que la réponse n'est pas un `nil`. Ci-dessous un petit exemple listant les module connectés

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

var
  module : TYModule;
  errmsg : string;

begin
  // Setup the API to use local USB devices
```

```
  if yRegisterHub('usb', errmsg)<>YAPI_SUCCESS then
  begin
    Write('RegisterHub error: '+errmsg);
    exit;
  end;

  Writeln('Device list');

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

end.
```

19.4. Gestion des erreurs

Lorsque vous implémentez un programme qui doit interagir avec des modules USB, vous ne pouvez pas faire abstraction de la gestion des erreurs. Il y aura forcément une occasion où un utilisateur aura débranché le périphérique, soit avant de lancer le programme, soit même en pleine opération. La librairie Yoctopuce est prévue pour vous aider à supporter ce genre de comportements, mais votre code doit néanmoins être fait pour se comporter au mieux pour interpréter les erreurs signalées par la librairie.

La manière la plus simple de contourner le problème est celle que nous avons employé pour les petits exemples précédents de ce chapitre: avant d'accéder à un module, on vérifie qu'il est en ligne avec la méthode `isOnline()` et on suppose ensuite qu'il va y rester pendant la fraction de seconde nécessaire à exécuter les lignes de code suivantes. Ce n'est pas parfait, mais ça peut suffire dans certains cas. Il faut toutefois être conscient qu'on ne peut pas totalement exclure une erreur se produisant après le `isOnline()`, qui pourrait faire planter le programme. La seule manière de l'éviter est d'implémenter une des deux techniques de gestion des erreurs décrites ci-dessous.

La méthode recommandée par la plupart des langages de programmation pour la gestion des erreurs imprévisibles est l'utilisation d'exceptions. C'est le comportement par défaut de la librairie Yoctopuce. Si une erreur se produit alors qu'on essaie d'accéder à un module, la librairie va lancer une exception. Dans ce cas, de trois choses l'une:

- Si votre code attrape l'exception au vol et la gère, et tout se passe bien.
- Si votre programme tourne dans le debugger, vous pourrez relativement facilement déterminer où le problème s'est produit, et voir le message explicatif lié à l'exception.
- Sinon... l'exception va crasher votre programme, boum!

Comme cette dernière situation n'est pas la plus souhaitable, la librairie Yoctopuce offre une autre alternative pour la gestion des erreurs, permettant de faire un programme robuste sans devoir attraper les exceptions à chaque ligne de code. Il suffit d'appeler la fonction `YAPI.DisableExceptions()` pour commuter la librairie dans un mode où les exceptions de chaque fonction sont systématiquement remplacées par des valeurs de retour particulières, qui peuvent être testées par l'appelant lorsque c'est pertinent. Le nom de la valeur de retour en cas d'erreur pour chaque fonction est systématiquement documenté dans la référence de la librairie. Il suit toujours la même logique: une méthode `get_state()` retournera une valeur `NomDeClasse.STATE_INVALID`, une méthode `get_currentValue` retournera une valeur `NomDeClasse.CURRENTVALUE_INVALID`, etc. Dans tous les cas, la valeur retournée sera du type attendu, et ne sera pas un pointeur nul qui risquerait de faire crasher votre programme. Au pire, si vous affichez la valeur sans la tester, elle sera hors du cadre attendu pour la valeur retournée. Dans le cas de fonctions qui ne retournent à priori pas d'information, la valeur de retour sera `YAPI.SUCCESS` si tout va bien, et un code d'erreur différent en cas d'échec.

Quand vous travaillez sans les exceptions, il est possible d'obtenir un code d'erreur et un message expliquant l'origine de l'erreur en le demandant à l'objet qui a retourné une erreur à l'aide des

méthodes `errType()` et `errMessage()`. Ce sont les même informations qui auraient été associées à l'exception si elles avaient été actives.

20. Utilisation du Yocto-SDI12 avec Universal Windows Platform

Universal Windows Platform, abrégé UWP, est n'est pas un langage à propremement parler mais une plate-forme logicielle créée par Micorosft. Cette platform permet d'executer un nouveau type d'applications : les application universelle Windows. Ces applicaiton peuvent fonctionner sur toutes les machines qui fonctione sous Windows 10. Cela comprend les PCs, les tablettes, les smartphones, la XBox One, mais aussi Windows IoT Core.

La librairie Yoctopuce UWP permet d'utiliser les modules Yoctopuce dans une application universelle Winodws et est entièrement écrite C#. Elle peut être ajoutée a un projet Visual Studio 2017[1].

20.1. Fonctions bloquantes et fonctions asynchrones

La librairie Universal Windows Platform n'utilise pas l'API win32 mais uniquement l'API Windows Runtime qui est disponible sur toutes les versions de Windows 10 et pour n'importe quelle architecture. Grâce à cela la librairie UWP peut être utilisé sur toutes les versions de Windows 10, y compris Windows 10 IoT Core.

Cependant, l'utilisation des nouvelles API UWP n'est pas sans conséquence: l'API Windows Runtime pour accéder aux ports USB est asynchrone, et par conséquent la librairie Yoctopuce doit aussi être asynchrone. Concrètement les méthodes asynchrones ne retournent pas directement le résultat mais un objet `Task` ou `Task<>` et le résultat peut être obtenu plus tard. Fort heureusement, le langage C# version 6 supporte les mots-clefs `async` et `await` qui simplifie beaucoup l'utilisation de ces fonctions. Il est ainsi possible d'utiliser les fonctions asynchrones de la même manière que les fonctions traditionnelles pour autant que les deux règles suivantes soient respectées:

- La méthode est déclarée comme asynchrone à l'aide du mot-clef `async`
- le mot-clef `await` est ajouté lors de l'utilisation d'une fonction asynchrone

Exemple:

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

    return result;
}
```

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

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

Notre librairie suit ces deux règles et peut donc dutiliser la notation `await`.

Pour ne pas devoir vous poser la question pour chaque méthode de savoir si elle est asynchrone ou pas, la convention est la suivante: **toutes les méthodes publiques** de la librairie UWP **sont asyncrones**, c'est-à-dire qui faut les appeler en ajoutant le mot clef `await`, **sauf**:

- `GetTickCount()`, parce que mesurer le temps de manière asynchrone n'a pas beaucoup de sens...
- `FindModule()`, `FirstModule()`, `nextModule()`, ... parce que la détection et l'énumération des modules est faite en tâche de fond sur des structures internes qui sont gérées de manière transparente, et qu'il n'est donc pas nécessaire de faire des opérations bloquantes durant le simple parcours de ces listes de modules.

20.2. Installation

Téléchargez la librairie Yoctopuce pour Universal Windows Platform depuis le site web de Yoctopuce [2]. Il n'y a pas de programme d'installation, copiez simplement de contenu du fichier zip dans le répertoire de votre choix. Vous avez besoin essentiellement du contenu du répertoire `Sources`. Les autres répertoires contiennent la documentation et quelques programmes d'exemple. Les projets d'exemple sont des projets Visual Studio 2017 qui est disponible sur le site de Microsoft [3].

20.3. Utilisation l'API Yoctopuce dans un projet Visual Studio

Commencez par créer votre projet , puis depuis le panneau **Explorateur de solutions** effectuez un clic droit sur votre projet, et choisissez **Ajouter** puis **Élément existant** .

Une fenêtre de sélection de fichiers apparaît: sélectionnez tous les fichiers du répertoire `Sources` de la librairie.

Vous avez alors le choix entre simplement ajouter ces fichiers à votre projet, ou les ajouter en tant que lien (le bouton **Ajouter** est en fait un menu déroulant). Dans le premier cas, Visual Studio va copier les fichiers choisis dans votre projet, dans le second Visual Studio va simplement garder un lien sur les fichiers originaux. Il est recommandé d'utiliser des liens, une éventuelle mise à jour de la librairie sera ainsi beaucoup plus facile.

Le fichier Package.appxmanifest

Par défaut, une application Universal Windows n'a pas le droit daccéder aux ports USB. Si l'on désire accéder à un périphérique USB, il faut impérativement le déclarer dans le fichier `Package.appxmanifest`.

Malheureusement, la fenêtre d'édition de ce fichier ne permet pas cette opération et il faut modifier le fichier `Package.appxmanifest` à la main. Dans le panneau "Solutions Explorer", faites un clic droit sur le fichier `Package.appxmanifest` et sélectionner "View Code".

Dans ce fichier XML, il faut rajouter un nud `DeviceCapability` dans le nud `Capabilities`. Ce nud doit avoir un attribut "Name" qui vaut "humaninterfacedevice".

A lintérieur de ce nud, il faut déclarer tous les modules qui peuvent être utilisés. Concrètement, pour chaque module, il faut ajouter un nud "Device" avec un attribut "Id" dont la valeur est une chaîne de caractères "vidpid:`USB_VENDORID USB_DEVICE_ID`". Le `USB_VENDORID` de Yoctopuce est 24e0 et le `USB_DEVICE_ID` de chaque module Yoctopuce peut être trouvé dans la

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

documentation dans la section "Caractéristiques". Pour finir, le n u d "Device" doit contenir un n u d "Function" avec l'attribut "Type" dont la valeur est "usage:ff00 0001".

Pour le Yocto-SDI12 voici ce qu'il faut ajouter dans le n u d "Capabilities":

```
<DeviceCapability Name="humaninterfacedevice">
    <!-- Yocto-SDI12 -->
    <Device Id="vidpid:24e0 00AB">
      <Function Type="usage:ff00 0001" />
    </Device>
  </DeviceCapability>
```

Malheureusement, il n'est pas possible d'écrire un règle qui autorise tous les modules Yoctopuce, par conséquent il faut impérativement ajouter chaque module que l'on désire utiliser.

20.4. Contrôle de la fonction Sdi12Port

Il suffit de quelques lignes de code pour piloter un Yocto-SDI12. Voici le squelette d'un fragment de code c# qui utilise la fonction Sdi12Port.

```
[...]
// On active la détection des modules sur USB
await YAPI.RegisterHub("usb");
[...]

// On récupère l'objet permettant d'intéragir avec le module
YSdi12Port sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.sdi12Port");

// Pour gérer le hot-plug, on vérifie que le module est là
if (await sdi12port.isOnline())
{
        // Use sdi12port.set_sdi12Mode()
    ...
}

[...]
```

Voyons maintenant en détail ce que font ces quelques lignes.

YAPI.RegisterHub

La fonction `YAPI.RegisterHub` initialise l'API de Yoctopuce en indiquant où les modules doivent être recherchés. Le paramètre est l'adresse du virtual hub capable de voir les modules. Si l'on passe la chaîne de caractère `"usb"`, l'API va travailler avec les modules connectés localement à la machine. Si l'initialisation se passe mal, une exception sera générée.

YSdi12Port.FindSdi12Port

La fonction `YSdi12Port.FindSdi12Port` permet de retrouver un port SDI12 en fonction du numéro de série de son module hôte et de son nom de fonction. Mais vous pouvez tout aussi bien utiliser des noms logiques que vous auriez préalablement configurés. Imaginons un module Yocto-SDI12 avec le numéros de série *YSDIMK01-123456* que vous auriez appelé "*MonModule*" et dont vous auriez nommé la fonction *sdi12Port* "*MaFonction*", les cinq appels suivants seront strictement équivalents (pour autant que *MaFonction* ne soit définie qu'une fois, pour éviter toute ambiguïté):

```
sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.sdi12Port");
sdi12port = YSdi12Port.FindSdi12Port("YSDIMK01-123456.MaFonction");
sdi12port = YSdi12Port.FindSdi12Port("MonModule.sdi12Port");
sdi12port = YSdi12Port.FindSdi12Port("MonModule.MaFonction");
sdi12port = YSdi12Port.FindSdi12Port("MaFonction");
```

`YSdi12Port.FindSdi12Port` renvoie un objet que vous pouvez ensuite utiliser à loisir pour contrôler le port SDI12.

isOnline

La méthode `isOnline()` de l'objet renvoyé par `YSdi12Port.FindSdi12Port` permet de savoir si le module correspondant est présent et en état de marche.

reset

La méthode `reset()` de l'objet returné par `YSdi12Port.FindSerialPort` vide tous les tampons du port série.

discoverSingleSensor

La méthode `discoverSingleSensor()` cherche l'adresse du capteur connecté sur le port SDI-12 et renvoie un objet avec toute les informations du capteur.

readSensor

La méthode `readSensor()` transmet la commande spécifiée sur le port SDI-12 au capteur spécifié et renvoie une liste d'objet avec toute les valeurs envoyées par le capteur.

20.5. Un exemple concret

Lancez Visual Studio et ouvrez le projet correspondant, fourni dans le répertoire **Examples/Doc-GettingStarted-Yocto-SDI12** de la librairie Yoctopuce.

Le projets Visual Studio contient de nombreux fichiers dont la plupart ne sont pas liés à l'utilisation de la librairie Yoctopuce. Pour simplifier la lecture du code nous avons regroupé tout le code qui utilise la librairie dans la classe `Demo` qui se trouve dans le fichier `demo.cs`. Les propriétés de cette classe correspondent aux différentes champs qui sont affichés à l'écran, et la méthode `Run()` contient le code qui est exécuté quand le bouton "Start" est pressé.

Vous reconnaîtrez dans cet exemple l'utilisation des fonctions expliquées ci-dessus, cette fois utilisées avec le décorum nécessaire à en faire un petit programme d'exemple concret.

```
using System;
using System.Collections.Generic;
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 Value { get; set; }

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

        YSdi12Port sdi12Port;

        int value = Convert.ToInt32(Value);

        if (Target.ToLower() == "any") {
          sdi12Port = YSdi12Port.FirstSdi12Port();
          if (sdi12Port == null) {
            WriteLine("No module connected (check USB cable) ");
            return -1;
          }

          Target = await (await sdi12Port.get_module()).get_serialNumber();
        }

        sdi12Port = YSdi12Port.FindSdi12Port(Target + ".sdi12Port");
        if (await sdi12Port.isOnline()) {
```

```
          YSdi12SensorInfo singleSensor = await sdi12Port.discoverSingleSensor();
          WriteLine("Module : " + Target);
          WriteLine("Sensor address : " + await singleSensor.get_sensorAddress()) ;
          WriteLine("Sensor SDI-12 compatibility : " + await
                    singleSensor.get_sensorProtocol());
          WriteLine("Sensor company name : " + await singleSensor.get_sensorVendor());
          WriteLine("Sensor model number : " + await singleSensor.get_sensorModel());
          WriteLine("Sensor version : " + await singleSensor.get_sensorVersion());
          WriteLine("Sensor serial number : " + await singleSensor.get_sensorSerial());
          List<double> valSensor = await sdi12Port.readSensor(await
                                   singleSensor.get_sensorAddress(), "M", 5000);

          for (int i = 0; i < valSensor.Count; i++) {
            if (await singleSensor .get_measureCount() > 1) {
              WriteLine(String.Format("{0} : {1,-8:0.00} {2,-10} ({3})",
                                      await singleSensor.get_measureSymbol(i), valSensor[i
],
                                      await singleSensor.get_measureUnit(i),
                                      await singleSensor.get_measureDescription(i)));
            } else {
              WriteLine(valSensor[i].ToString());
            }
          }

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

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

20.6. Contrôle de la partie module

Chaque module peut-être contrôlé d'une manière similaire, vous trouverez ci-dessous un simple programme d'exemple affichant les principaux paramètres d'un module et permettant d'activer la balise de localisation.

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

namespace Demo
{
  public class Demo : DemoBase
  {

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

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

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

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

Chaque propriété `xxx` du module peut être lue grâce à une méthode du type `YModule.get_xxxx()`, et les propriétés qui se sont pas en lecture seule peuvent être modifiées à l'aide de la méthode `YModule.set_xxx()` Pour plus de détails concernant ces fonctions utilisées, reportez-vous aux chapitre API

Modifications des réglages du module

Lorsque que vous souhaitez modifier les réglages d'un module, il suffit d'appeler la fonction `YModule.set_xxx()` correspondante, cependant cette modification n'a lieu que dans la mémoire vive du module: si le module redémarre, les modifications seront perdues. Pour qu'elle soient mémorisées de manière persistante, il est nécessaire de demander au module de sauvegarder sa configuration courante dans sa mémoire non volatile. Pour cela il faut utiliser la méthode `YModule.saveToFlash()`. Inversement il est possible de forcer le module à oublier ses réglages courants en utilisant la méthode `YModule.revertFromFlash()`. Ce petit exemple ci-dessous vous permet changer le nom logique d'un 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;
    }
  }
}
```

Attention, le nombre de cycles d'écriture de la mémoire non volatile du module est limité. Passé cette limite plus rien ne garantit que la sauvegarde des réglages se passera correctement. Cette limite, liée à la technologie employée par le micro-processeur du module se situe aux alentour de 100000 cycles. Pour résumer vous ne pouvez employer la fonction `YModule.saveToFlash()` que 100000 fois au cours de la vie du module. Veillez donc à ne pas appeler cette fonction depuis l'intérieur d'une boucle.

Enumeration des modules

Obtenir la liste des modules connectés se fait à l'aide de la fonction `YModule.yFirstModule()` qui renvoie le premier module trouvé, il suffit ensuite d'appeler la méthode `nextModule()` de cet objet pour trouver les modules suivants, et ce tant que la réponse n'est pas un `null`. Ci-dessous un petit exemple listant les module connectés

```
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;
    }
  }
}
```

20.7. Gestion des erreurs

Lorsque vous implémentez un programme qui doit interagir avec des modules USB, vous ne pouvez pas faire abstraction de la gestion des erreurs. Il y aura forcément une occasion où un utilisateur aura débranché le périphérique, soit avant de lancer le programme, soit même en pleine opération. La librairie Yoctopuce est prévue pour vous aider à supporter ce genre de comportements, mais votre code doit néanmoins être fait pour se comporter au mieux pour interpréter les erreurs signalées par la librairie.

La manière la plus simple de contourner le problème est celle que nous avons employé pour les petits exemples précédents de ce chapitre: avant d'accéder à un module, on vérifie qu'il est en ligne avec la méthode `isOnline()` et on suppose ensuite qu'il va y rester pendant la fraction de

seconde nécessaire à exécuter les lignes de code suivantes. Ce n'est pas parfait, mais ça peut suffire dans certains cas. Il faut toutefois être conscient qu'on ne peut pas totalement exclure une erreur se produisant après le `isOnline()`, qui pourrait faire planter le programme.

Dans la librairie Universal Windows Platform, le traitement d'erreur est implémenté au moyen d'exceptions. Vous devrez donc intercepter et traiter correctement ces exceptions si vous souhaitez avoir un projet fiable qui ne crashera pas des que vous débrancherez un module.

Les exceptions lancées de la librairie sont toujours de type YAPI_Exception, ce qui permet facilement de les séparer des autres exceptions dans un bloc `try{...} catch{...}`.

Exemple:

```
try {
        ....
} catch (YAPI_Exception ex) {
        Debug.WriteLine("Exception from Yoctopuce lib:" + ex.Message);
} catch (Exception ex) {
        Debug.WriteLine("Other exceptions :" + ex.Message);
}
```

21. Utilisation du Yocto-SDI12 en Objective-C

Objective-C est le langage de prédilection pour programmer sous macOS, en raison de son intégration avec le générateur d'interfaces Cocoa. Yoctopuce supporte les versions de XCode supportées par Apple. La librairie Yoctopuce est compatible ARC. Il vous sera donc possible de coder vos projet soit en utilisant la traditionnelle méthode de *retain / release*, soit en activant l'*Automatic Reference Counting*.

Les librairies Yoctopuce[1] pour Objective-C vous sont fournies au format source dans leur intégralité. Une partie de la librairie de bas-niveau est écrite en C pur sucre, mais vous n'aurez à priori pas besoin d'interagir directement avec elle: tout a été fait pour que l'interaction soit le plus simple possible depuis Objective-C.

Vous allez rapidement vous rendre compte que l'API Objective-C définit beaucoup de fonctions qui retournent des objets. Vous ne devez jamais désallouer ces objets vous-même. Ils seront désalloués automatiquement par l'API à la fin de l'application.

Afin des les garder simples, tous les exemples fournis dans cette documentation sont des applications consoles. Il va de soit que que les fonctionnement des librairies est strictement identiques si vous les intégrez dans une application dotée d'une interface graphique. Vous trouverez sur le blog de Yoctopuce un exemple détaillé[2] avec des séquences vidéo montrant comment intégrer les fichiers de la librairie à vos projets.

21.1. Contrôle de la fonction Sdi12Port

Il suffit de quelques lignes de code pour piloter un Yocto-SDI12. Voici le squelette d'un fragment de code Objective-C qui utilise la fonction Sdi12Port.

```
#import "yocto_api.h"
#import "yocto_sdi12port.h"

...
NSError *error;
[YAPI RegisterHub:@"usb": &error]
...
// On récupère l'objet représentant le module (ici connecté en local sur USB)
sdi12port = [YSdi12Port FindSdi12Port:@"YSDIMK01-123456.sdi12Port"];

// Pour gérer le hot-plug, on vérifie que le module est là
if([sdi12port isOnline])
```

[1] www.yoctopuce.com/FR/libraries.php
[2] www.yoctopuce.com/FR/article/nouvelle-librairie-objective-c-pour-mac-os-x

```
{
    // Utiliser [sdi12port set_sdi12Mode]
    ...
}
```

Voyons maintenant en détail ce que font ces quelques lignes.

yocto_api.h et yocto_sdi12port.h

Ces deux fichiers importés permettent d'avoir accès aux fonctions permettant de gérer les modules Yoctopuce. `yocto_api.h` doit toujours être utilisé, `yocto_sdi12port.h` est nécessaire pour gérer les modules contenant un port SDI12, comme le Yocto-SDI12.

[YAPI RegisterHub]

La fonction `[YAPI RegisterHub]` initialise l'API de Yoctopuce en indiquant où les modules doivent être recherchés. Utilisée avec le paramètre `@"usb"`, elle permet de travailler avec les modules connectés localement à la machine. Si l'initialisation se passe mal, cette fonction renverra une valeur différente de `YAPI_SUCCESS`, et retournera via le paramètre `errmsg` un explication du problème.

[Sdi12Port FindSdi12Port]

La fonction `[Sdi12Port FindSdi12Port]`, permet de retrouver un port SDI12 en fonction du numéro de série de son module hôte et de son nom de fonction. Mais vous pouvez tout aussi bien utiliser des noms logiques que vous auriez préalablement configurés. Imaginons un module Yocto-SDI12 avec le numéros de série *YSDIMK01-123456* que vous auriez appelé "*MonModule*" et dont vous auriez nommé la fonction *sdi12Port* "*MaFonction*", les cinq appels suivants seront strictement équivalents (pour autant que *MaFonction* ne soit définie qu'une fois, pour éviter toute ambiguïté):

```
YSdi12Port *sdi12port = [YSdi12Port FindSdi12Port:@"YSDIMK01-123456.sdi12Port"];
YSdi12Port *sdi12port = [YSdi12Port FindSdi12Port:@"YSDIMK01-123456.MaFonction"];
YSdi12Port *sdi12port = [YSdi12Port FindSdi12Port:@"MonModule.sdi12Port"];
YSdi12Port *sdi12port = [YSdi12Port FindSdi12Port:@"MonModule.MaFonction"];
YSdi12Port *sdi12port = [YSdi12Port FindSdi12Port:@"MaFonction"];
```

`[YSdi12Port FindSdi12Port]` renvoie un objet que vous pouvez ensuite utiliser à loisir pour contrôler le port SDI12.

isOnline

La méthode `isOnline` de l'objet renvoyé par `[YSdi12Port FindSdi12Port]` permet de savoir si le module correspondant est présent et en état de marche.

reset

La méthode `reset()` de l'objet returné par `YSdi12Port.FindSerialPort` vide tous les tampons du port série.

discoverSingleSensor

La méthode `discoverSingleSensor()` cherche l'adresse du capteur connecté sur le port SDI-12 et renvoie un objet avec toute les informations du capteur.

readSensor

La méthode `readSensor()` transmet la commande spécifiée sur le port SDI-12 au capteur spécifié et renvoie une liste d'objet avec toute les valeurs envoyées par le capteur.

Un exemple réel

Lancez Xcode 4.2 et ouvrez le projet exemple correspondant, fourni dans le répertoire **Examples/Doc-GettingStarted-Yocto-SDI12** de la librairie Yoctopuce.

Vous reconnaîtrez dans cet exemple l'utilisation des fonctions expliquées ci-dessus, cette fois utilisées avec le décorum nécessaire à en faire un petit programme d'exemple concret.

```
#import <Foundation/Foundation.h>
#import "yocto_api.h"
#import "yocto_sdi12port.h"

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

  @autoreleasepool {
    NSError *error;
    // Setup the API to use local USB devices
    if([YAPI RegisterHub:@"usb": &error] != YAPI_SUCCESS) {
      NSLog(@"RegisterHub error: %@", [error localizedDescription]);
      return 1;
    }
    YSdi12Port *sdi12port;
    if (argc > 1) {
      NSString     *target = [NSString stringWithUTF8String:argv[1]];
      sdi12port = [YSdi12Port FindSdi12Port:target];
    } else {
      sdi12port = [YSdi12Port FirstSdi12Port];
      if (sdi12port == NULL) {
        NSLog(@"No module connected (check USB cable)");
        return 1;
      }
    }
    [sdi12port reset];
    YSdi12SensorInfo *singleSensor = [sdi12port discoverSingleSensor];
    NSLog(@"Sensor address : %@", [singleSensor get_sensorAddress]);
    NSLog(@"Sensor SDI-12 compatibility : %@", [singleSensor get_sensorProtocol]);
    NSLog(@"Sensor company name : %@", [singleSensor get_sensorVendor]);
    NSLog(@"Sensor model number : %@", [singleSensor get_sensorModel]);
    NSLog(@"Sensor version : %@", [singleSensor get_sensorVersion]);
    NSLog(@"Sensor serial number : %@", [singleSensor get_sensorSerial]);
    NSMutableArray* valSensor = [sdi12port  readSensor:[singleSensor get_sensorAddress] :
                                 @"M" :5000];
    for (int i = 0 ; i < [valSensor count]; i++) {
      if ([singleSensor get_measureCount] > 1) {
        NSLog(@"%@ %f%@ (%@) \n", [singleSensor get_measureSymbol :i], [[valSensor
objectAtIndex:
              i] doubleValue],
              [singleSensor get_measureUnit :i], [singleSensor get_measureDescription :i]);
      } else {
        NSLog(@"%f", [[valSensor objectAtIndex:i] doubleValue]);
      }
    }
    [YAPI FreeAPI];
  }
  return 0;
}
```

21.2. Contrôle de la partie module

Chaque module peut-être contrôlé d'une manière similaire, vous trouverez ci dessous un simple programme d'exemple affichant les principaux paramètres d'un module et permettant d'activer la balise de localisation.

```
#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;
}
```

Chaque propriété `xxx` du module peut être lue grâce à une méthode du type `get_xxxx`, et les propriétés qui se sont pas en lecture seule peuvent être modifiées à l'aide de la méthode `set_xxx:` Pour plus de détails concernant ces fonctions utilisées, reportez-vous aux chapitre API

Modifications des réglages du module

Lorsque que vous souhaitez modifier les réglages d'un module, il suffit d'appeler la fonction `set_xxx:` correspondante, cependant cette modification n'a lieu que dans la mémoire vive du module: si le module redémarre, les modifications seront perdues. Pour qu'elle soient mémorisées de manière persistante, il est nécessaire de demander au module de sauvegarder sa configuration courante dans sa mémoire non volatile. Pour cela il faut utiliser la méthode `saveToFlash`. Inversement il est possible de forcer le module à oublier ses réglages courants en utilisant la méthode `revertFromFlash`. Ce petit exemple ci-dessous vous permet changer le nom logique d'un 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;
}
```

Attention, le nombre de cycles d'écriture de la mémoire non volatile du module est limité. Passé cette limite plus rien ne garantit que la sauvegarde des réglages se passera correctement. Cette limite, liée à la technologie employée par le micro-processeur du module se situe aux alentour de 100000 cycles. Pour résumer vous ne pouvez employer la fonction `saveToFlash` que 100000 fois au cours de la vie du module. Veillez donc à ne pas appeler cette fonction depuis l'intérieur d'une boucle.

Enumeration des modules

Obtenir la liste des modules connectés se fait à l'aide de la fonction `yFirstModule()` qui renvoie le premier module trouvé, il suffit ensuite d'appeler la fonction `nextModule()` de cet objet pour trouver les modules suivants, et ce tant que la réponse n'est pas un `NULL`. Ci-dessous un petit exemple listant les module connectés

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

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

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

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

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

21.3. Gestion des erreurs

Lorsque vous implémentez un programme qui doit interagir avec des modules USB, vous ne pouvez pas faire abstraction de la gestion des erreurs. Il y aura forcément une occasion où un utilisateur

aura débranché le périphérique, soit avant de lancer le programme, soit même en pleine opération. La librairie Yoctopuce est prévue pour vous aider à supporter ce genre de comportements, mais votre code doit néanmoins être fait pour se comporter au mieux pour interpréter les erreurs signalées par la librairie.

La manière la plus simple de contourner le problème est celle que nous avons employé pour les petits exemples précédents de ce chapitre: avant d'accéder à un module, on vérifie qu'il est en ligne avec la méthode `isOnline()` et on suppose ensuite qu'il va y rester pendant la fraction de seconde nécessaire à exécuter les lignes de code suivantes. Ce n'est pas parfait, mais ça peut suffire dans certains cas. Il faut toutefois être conscient qu'on ne peut pas totalement exclure une erreur se produisant après le `isOnline()`, qui pourrait faire planter le programme. La seule manière de l'éviter est d'implémenter une des deux techniques de gestion des erreurs décrites ci-dessous.

La méthode recommandée par la plupart des langages de programmation pour la gestion des erreurs imprévisibles est l'utilisation d'exceptions. C'est le comportement par défaut de la librairie Yoctopuce. Si une erreur se produit alors qu'on essaie d'accéder à un module, la librairie va lancer une exception. Dans ce cas, de trois choses l'une:

- Si votre code attrape l'exception au vol et la gère, et tout se passe bien.
- Si votre programme tourne dans le debugger, vous pourrez relativement facilement déterminer où le problème s'est produit, et voir le message explicatif lié à l'exception.
- Sinon... l'exception va crasher votre programme, boum!

Comme cette dernière situation n'est pas la plus souhaitable, la librairie Yoctopuce offre une autre alternative pour la gestion des erreurs, permettant de faire un programme robuste sans devoir attraper les exceptions à chaque ligne de code. Il suffit d'appeler la fonction `YAPI.DisableExceptions()` pour commuter la librairie dans un mode où les exceptions de chaque fonction sont systématiquement remplacées par des valeurs de retour particulières, qui peuvent être testées par l'appelant lorsque c'est pertinent. Le nom de la valeur de retour en cas d'erreur pour chaque fonction est systématiquement documenté dans la référence de la librairie. Il suit toujours la même logique: une méthode `get_state()` retournera une valeur `NomDeClasse.STATE_INVALID`, une méthode `get_currentValue` retournera une valeur `NomDeClasse.CURRENTVALUE_INVALID`, etc. Dans tous les cas, la valeur retournée sera du type attendu, et ne sera pas un pointeur nul qui risquerait de faire crasher votre programme. Au pire, si vous affichez la valeur sans la tester, elle sera hors du cadre attendu pour la valeur retournée. Dans le cas de fonctions qui ne retournent à priori pas d'information, la valeur de retour sera `YAPI.SUCCESS` si tout va bien, et un code d'erreur différent en cas d'échec.

Quand vous travaillez sans les exceptions, il est possible d'obtenir un code d'erreur et un message expliquant l'origine de l'erreur en le demandant à l'objet qui a retourné une erreur à l'aide des méthodes `errType()` et `errMessage()`. Ce sont les même informations qui auraient été associées à l'exception si elles avaient été actives.

22. Utilisation avec des langages non supportés

Les modules Yoctopuce peuvent être contrôlés depuis la plupart des langages de programmation courants. De nouveaux langages sont ajoutés régulièrement en fonction de l'intérêt exprimé par les utilisateurs de produits Yoctopuce. Cependant, certains langages ne sont pas et ne seront jamais supportés par Yoctopuce, les raisons peuvent être diverses: compilateurs plus disponibles, environnements inadaptés, etc...

Il existe cependant des méthodes alternatives pour accéder à des modules Yoctopuce depuis un langage de programmation non supporté.

22.1. Utilisation en ligne de commande

Le moyen le plus simple pour contrôler des modules Yoctopuce depuis un langage non supporté consiste à utiliser l'API en ligne de commande à travers des appels système. L'API en ligne de commande se présente en effet sous la forme d'un ensemble de petits exécutables qu'il est facile d'appeler et dont la sortie est facile à analyser. La plupart des langages de programmation permettant d'effectuer des appels système, cela permet de résoudre le problème en quelques lignes.

Cependant, si l'API en ligne de commande est la solution la plus facile, ce n'est pas la plus rapide ni la plus efficace. A chaque appel, l'exécutable devra initialiser sa propre API et faire l'inventaire des modules USB connectés. Il faut compter environ une seconde par appel.

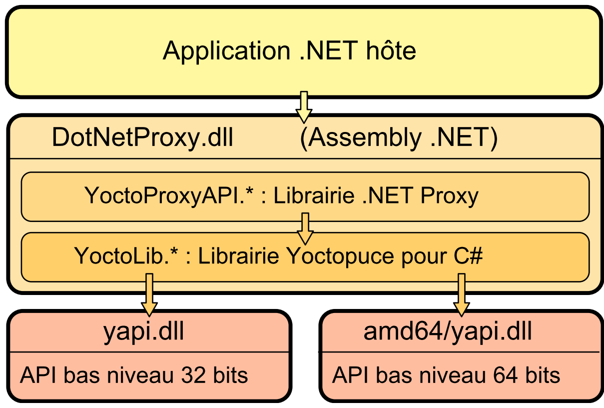
22.2. Assembly .NET

Un Assembly .NET permet de partager un ensemble de classes précompilées pour offrir un service, en annonçant des points d'entrées qui peuvent être utilisés par des applications tierces. Dans notre cas, c'est toute la librairie Yoctopuce qui est disponible dans l'Assembly .NET, de sorte à pouvoir être utilisée dans n'importe quel environnement qui supporte le chargement dynamique d'Assembly .NET.

La librairie Yoctopuce sous forme d'Assembly .NET ne contient pas uniquement la librairie Yoctopuce standard pour C#, car cela n'aurait pas permis une utilisation optimale dans tous les environnements. En effet, on ne peut pas attendre forcément des applications hôtes d'offrir un système de threads ou de callbacks, pourtant très utiles pour la gestion du plug-and-play et des capteurs à taux de rafraîchissements élevé. De même, on ne peut pas attendre des applications externes un comportement transparent dans le cas où un appel de fonction dans l'Assembly cause un délai en raison de communication réseau.

Nous y avons donc ajouté une surcouche, appelée librairie *.NET Proxy*. Cette surcouche offre une interface très similaire à la librairie standard mais un peu simplifiée, car elle gère en interne tous les

mécanismes de callbacks. A la place, cette librairie offre des objets miroirs, appelés *Proxys*, qui publient par le biais de *Propriétés* les principaux attributs des fonctions Yoctopuce tels que la mesure courante, les paramètres de configuration, l'état, etc.



Architecture de l'Assembly .NET

Les propriétés des objets *Proxys* sont automatiquement mises à jour en tâche de fond par le mécanisme de callbacks, sans que l'application hôte n'ait à s'en soucier. Celle-ci peut donc à tout moment et sans aucun risque de latence afficher la valeur de toutes les propriétés des objets Proxys Yoctopuce.

Notez bien que la librairie de communication de bas niveau `yapi.dll` n'est **pas** inclue dans l'Assembly .NET. Il faut donc bien penser à la garder toujours avec `DotNetProxyLibrary.dll`. La version 32 bits doit être dans le même répertoire que `DotNetProxyLibrary.dll`, tandis que la version 64 bits doit être dans un sous-répertoire nommé `amd64`.

Exemple d'utilisation avec MATLAB

Voici comment charger notre Assembly .NET Proxy dans MATLAB et lire la valeur du premier capteur branché par USB trouvé sur la machine :

```
NET.addAssembly("C:/Yoctopuce/DotNetProxyLibrary.dll");
import YoctoProxyAPI.*

errmsg = YAPIProxy.RegisterHub("usb");
sensor = YSensorProxy.FindSensor("");
measure = sprintf('%.3f %s', sensor.CurrentValue, sensor.Unit);
```

Exemple d'utilisation en PowerShell

Les commandes en PowerShell sont un peu plus étranges, mais on reconnaît le même schéma :

```
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
```

Particularités de la librairie .NET Proxy

Par rapport aux librairies Yoctopuce classiques, on notera en particulier les différences suivantes.

Pas de méthode FirstModule/nextModule

Pour obtenir un objet se référant au premier module trouvé, on appelle un `YModuleProxy.FindModule("")`. Si aucun module n'est connecté, cette méthode retournera un objet avec la propriété `module.IsOnline` à `False`. Dès le branchement d'un module, la propriété passera à `True` et l'identifiant matériel du module sera mis à jour.

Pour énumérer les modules, on peut appeler la méthode `module.GetSimilarFunctions()` qui retourne un tableau de chaînes de caractères contenant les identifiants de tous les module trouvés.

Pas de fonctions de callback
Les fonctions de callback sont implémentées en interne et mettent à jour les propriétés des objets. Vous pouvez donc simplement faire du polling sur les propriétés, sans pénalité significative de performance. Prenez garde au fait que si vous utilisez l'une des méthodes qui désactive les callbacks, le rafraichissement automatique des propriétés des objets en sera altéré.

Une nouvelle méthode `YAPIProxy.GetLog` permet de récupérer les logs de diagnostiques de bas niveau sans recourir à l'utilisation de callbacks.

Types énumérés
Pour maximiser la compatibilité avec les applications hôte, la librairie .NET Proxy n'utilise pas de véritables types énumérés .NET, mais des simples entiers. Pour chaque type énuméré, la librairie publie des constantes entières nommées correspondant aux valeurs possibles. Contrairement aux librairies Yoctopuce classiques, les valeurs utiles commencent toujours à `1`, la valeur `0` étant réservée pour signifier une valeur invalide, par exemple lorsque le module est débranché.

Valeurs numériques invalides
Pour toutes les grandeurs numériques, plutôt qu'une constante arbitraire, la valeur invalide retournée en cas d'erreur est *NaN*. Il faut donc utiliser la fonction *isNaN()* pour détecter cette valeur.

Utilisation de l'Assembly .NET sans la librairie Proxy

Si pour une raison ou une autre vous ne désirez pas utiliser la librairie Proxy, et que votre environnement le permet, vous pouvez utiliser l'API C# standard puisqu'elle se trouve dans l'Assembly, sous le namespace `YoctoLib`. Attention toutefois à ne pas mélanger les deux utilisations: soit vous passez par la librairie Proxy, soit vous utilisez directement la version `YoctoLib`, mais pas les deux !

Compatibilité

Pour que la librairie *.NET Proxy* fonctionne correctement avec vos modules Yoctopuce, ces derniers doivent avoir au moins le firmware 37120.

Afin d'être compatible avec un maximum de version de Windows, y compris Windows XP, la librairie *DotNetProxyLibrary.dll* est compilée en .NET 3.5, qui est disponible par défaut sur toutes les versions de Windows depuis XP. A ce jour nous n'avons pas trouvé d'environnement hormis Windows qui supporte le chargement d'Assemblys, donc seules les dll de bas niveau pour Windows sont distribuées avec l'Assembly.

22.3. Virtual Hub et HTTP GET

Le *Virtual Hub* est disponible pour presque toutes les plateformes actuelles, il sert généralement de passerelle pour permettre l'accès aux modules Yoctopuce depuis des langages qui interdisent l'accès direct aux couches matérielles d'un ordinateur (Javascript, PHP, Java...).

Il se trouve que le *Virtual Hub* est en fait un petit serveur Web qui est capable de router des requêtes HTTP vers les modules Yoctopuce. Ce qui signifie que si vous pouvez faire une requête HTTP depuis votre langage de programmation, vous pouvez contrôler des modules Yoctopuce, même si ce langage n'est pas officiellement supporté.

Interface REST

A bas niveau, les modules sont pilotés à l'aide d'une API REST. Ainsi pour contrôler un module, il suffit de faire les requêtes HTTP appropriées sur le *Virtual Hub*. Par défaut le port HTTP du *Virtual Hub* est 4444.

Un des gros avantages de cette technique est que les tests préliminaires sont très faciles à mettre en uvre, il suffit d'un *Virtual Hub* et d'un simple browser Web. Ainsi, si vous copiez l'URL suivante dans votre browser favori, alors que le *Virtual Hub* est en train de tourner, vous obtiendrez la liste des modules présents.

```
http://127.0.0.1:4444/api/services/whitePages.txt
```

Remarquez que le résultat est présenté sous forme texte, mais en demandant *whitePages.xml* vous auriez obtenu le résultat en XML. De même, *whitePages.json* aurait permis d'obtenir le résultat en JSON. L'extension *html* vous permet même d'afficher une interface sommaire vous permettant de changer les valeurs en direct. Toute l'API REST est disponible dans ces différents formats.

Contrôle d'un module par l'interface REST

Chaque module Yoctopuce a sa propre interface REST disponible sous différentes formes. Imaginons un Yocto-SDI12 avec le numéro de de série *YSDIMK01-12345* et le nom logique *monModule*. l'URL suivante permettra de connaître l'état du module.

```
http://127.0.0.1:4444/bySerial/YSDIMK01-12345/api/module.txt
```

Il est bien entendu possible d'utiliser le nom logique des modules plutôt que leur numéro de série.

```
http://127.0.0.1:4444/byName/monModule/api/module.txt
```

Vous pouvez retrouver la valeur d'une des propriétés d'un module, il suffit d'ajouter le nom de la propriété en dessous de *module*. Par exemple, si vous souhaitez connaître la luminosité des LEDs de signalisation, il vous suffit de faire la requête suivante:

```
http://127.0.0.1:4444/bySerial/YSDIMK01-12345/api/module/luminosity
```

Pour modifier la valeur d'une propriété, il vous suffit de modifier l'attribut correspondant. Ainsi, pour modifier la luminosité il vous suffit de faire la requête suivante:

```
http://127.0.0.1:4444/bySerial/YSDIMK01-12345/api/module?luminosity=100
```

Contrôle des différentes fonctions du module par l'interface REST

Les fonctionnalités des modules se manipulent de la même manière. Pour connaître l'état de la fonction sdi12Port, il suffit de construire l'URL suivante.

```
http://127.0.0.1:4444/bySerial/YSDIMK01-12345/api/sdi12Port.txt
```

En revanche, si vous pouvez utiliser le nom logique du module en lieu et place de son numéro de série, vous ne pouvez pas utiliser les noms logiques des fonctions, seuls les noms hardware sont autorisés pour les fonctions.

Vous pouvez retrouver un attribut d'une fonction d'un module d'une manière assez similaire à celle utilisée avec les modules, par exemple:

```
http://127.0.0.1:4444/bySerial/YSDIMK01-12345/api/sdi12Port/logicalName
```

Assez logiquement, les attributs peuvent être modifiés de la même manière.

```
http://127.0.0.1:4444/bySerial/YSDIMK01-12345/api/sdi12Port?logicalName=maFonction
```

Vous trouverez la liste des attributs disponibles pour votre Yocto-SDI12 au début du chapitre *Programmation, concepts généraux*.

Accès aux données enregistrées sur le datalogger par l'interface REST

Cette section s'applique uniquement aux modules dotés d'un enregistreur de donnée.

La version résumée des données enregistrées dans le datalogger peut être obtenue au format JSON à l'aide de l'URL suivante:

```
http://127.0.0.1:4444/bySerial/YSDIMK01-12345/dataLogger.json
```

Le détail de chaque mesure pour un chaque tranche d'enregistrement peut être obtenu en ajoutant à l'URL l'identifiant de la fonction désirée et l'heure de départ de la tranche:

```
http://127.0.0.1:4444/bySerial/YSDIMK01-12345/dataLogger.json?id=sdi12Port&utc=1389801080
```

22.4. Utilisation des librairies dynamiques

L'API Yoctopuce bas niveau est disponible sous différents formats de librairie dynamiques écrites en C, dont les sources sont disponibles avec l'API C++. Utiliser une de ces librairies bas niveau vous permettra de vous passer du *Virtual Hub*.

Filename	Plateforme
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)

Ces librairies dynamiques contiennent toutes les fonctionnalités nécessaires pour reconstruire entièrement toute l'API haut niveau dans n'importe quel langage capable d'intégrer ces librairies. Ce chapitre se limite cependant à décrire une utilisation de base des modules.

Contrôle d'un module

Les trois fonctions essentielles de l'API bas niveau sont les suivantes:

```
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);
```

La fonction *yapiInitAPI* permet d'initialiser l'API et doit être appelée une fois en début du programme. Pour une connection de type USB, le paramètre *connection_type* doit prendre la valeur 1. *errmsg* est un pointeur sur un buffer de 255 caractères destiné à récupérer un éventuel message d'erreur. Ce pointeur peut être aussi mis à *NULL*. La fonction retourne un entier négatif en cas d'erreur, ou zéro dans le cas contraire.

La fonction *yapiUpdateDeviceList* gère l'inventaire des modules Yoctopuce connectés, elle doit être appelée au moins une fois. Pour pouvoir gérer le hot plug, et détecter d'éventuels nouveaux modules connectés, cette fonction devra être apellée à intervalles réguliers. Le paramètre *forceupdate* devra être à la valeur 1 pour forcer un scan matériel. Le paramètre *errmsg* devra pointer sur un buffer de 255 caractères pour récupérer un éventuel message d'erreur. Ce pointeur peut aussi être à *null*.Cette fonction retourne un entier négatif en cas d'erreur, ou zéro dans le cas contraire.

Enfin, la fonction *yapiHTTPRequest* permet d'envoyer des requêtes HTTP à l'API REST du module. Le paramètre *device* devra contenir le numéro de série ou le nom logique du module que vous cherchez à atteindre. Le paramètre *request* doit contenir la requête HTTP complète (y compris les sauts de ligne terminaux). *buffer* doit pointer sur un buffer de caractères suffisamment grand pour

contenir la réponse. *buffsize* doit contenir la taille du buffer. *fullsize* est un pointeur sur un entier qui sera affecté à la taille effective de la réponse. Le paramètre *errmsg* devra pointer sur un buffer de 255 caractères pour récupérer un éventuel message d'erreur. Ce pointeur peut aussi être à *null*. Cette fonction retourne un entier négatif en cas d'erreur, ou zéro dans le cas contraire.

Le format des requêtes est le même que celui décrit dans la section *Virtual Hub et HTTP GET*. Toutes les chaînes de caractères utilisées par l'API sont des chaînes constituées de caractères 8 bits: l'Unicode et l'UTF8 ne sont pas supportés.

Le résultat retourné dans la variable buffer respecte le protocole HTTP, il inclut donc un header HTTP . Ce header se termine par deux lignes vides, c'est-à-dire une séquence de quatre caractères ASCII 13, 10, 13, 10.

Voici un programme d'exemple écrit en pascal qui utilise la DLL *yapi.dll* pour lire puis changer la luminosité d'un 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      = 'YSDIMK01-12345';
  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));

  // change the luminosity
  if (yapiHTTPRequest(serial,setValue,data,sizeof(dataBuffer),fullsize,errmsg)<0) then
   begin
     writeln(errmsg);
     halt;
```

```
   end;

end.
```

Inventaire des modules

Pour procéder à l'inventaire des modules Yoctopuce, deux fonctions de la librairie dynamique sont nécessaires

```
int yapiGetAllDevices(int *buffer,int maxsize,int *neededsize,char *errmsg);
int yapiGetDeviceInfo(int devdesc,yDeviceSt *infos, char *errmsg);
```

La fonction *yapiGetAllDevices* permet d'obtenir la liste des modules connectés sous la forme d'une liste de handles. *buffer* pointe sur un tableau d'entiers 32 bits qui contiendra les handles retournés. *Maxsize* est la taille en bytes du buffer. *neededsize* contiendra au retour la taille nécessaire pour stocker tous les handles. Cela permet d'en déduire le nombre de module connectés, ou si le buffer passé en entrée est trop petit. Le paramètre *errmsg* devra pointer sur un buffer de 255 caractères pour récupérer un éventuel message d'erreur. Ce pointeur peut aussi être à *null*. Cette fonction retourne un entier négatif en cas d'erreur, ou zéro dans le cas contraire.

La fonction *yapiGetDeviceInfo* permet de récupérer les informations relatives à un module à partir de son handle. *devdesc* est un entier 32bit qui représente le module, et qui a été obtenu grâce à *yapiGetAllDevices*. *infos* pointe sur une structure de données dans laquelle sera stocké le résultat. Le format de cette structure est le suivant:

Nom	Type	Taille (bytes)	Description
vendorid	int	4	ID USB de Yoctopuce
deviceid	int	4	ID USB du module
devrelease	int	4	Version du module
nbinbterfaces	int	4	Nombre d'interfaces USB utilisée par le module
manufacturer	char[]	20	Yoctopuce (null terminé)
productname	char[]	28	Modèle (null terminé)
serial	char[]	20	Numéro de série (null terminé)
logicalname	char[]	20	Nom logique (null terminé)
firmware	char[]	22	Version du firmware (null terminé)
beacon	byte	1	Etat de la balise de localisation (0/1)

Le paramètre *errmsg* devra pointer sur un buffer de 255 caractères pour récupérer un éventuel message d'erreur.

Voici un programme d'exemple écrit en pascal qui utilise la DLL *yapi.dll* pour lister les modules connectés.

```
// 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  initialisation
  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 et yapi.dll

Chaque point d'entrée de la DLL yapi.dll est disponible en deux versions, une classique C-decl, et un seconde compatible avec Visual Basic 6 préfixée avec *vb6_*.

22.5. Port de la librairie haut niveau

Toutes les sources de l'API Yoctopuce étant fournies dans leur intégralité, vous pouvez parfaitement entreprendre le port complet de l'API dans le langage de votre choix. Sachez cependant qu'une grande partie du code source de l'API est généré automatiquement.

Ainsi, il n'est pas nécessaire de porter la totalité de l'API, il suffit de porter le fichier *yocto_api* et un de ceux correspondant à une fonctionnalité, par exemple *yocto_relay*. Moyennant un peu de travail supplémentaire, Yoctopuce sera alors en mesure de générer tous les autres fichiers. C'est pourquoi il est fortement recommandé de contacter le support Yoctopuce avant d'entreprendre le port de la librairie Yoctopuce dans un autre langage. Un travail collaboratif sera profitable aux deux parties.

23. Programmation avancée

Les chapitres précédents vous ont présenté dans chaque language disponible les fonctions de programmation de base utilisables avec votre module Yocto-SDI12. Ce chapitre présente de façon plus générale une utilisation plus avancée de votre module. Les exemples sont donnés dans le language le plus populaire auprès des clients de Yoctopuce, à savoir C#. Néanmoins, vous trouverez dans les librairies de programmation pour chaque language des exemples complets illustrant les concepts présentés ici.

Afin de rester le plus concis possible, les exemples donnés dans ce chapitre ne font aucune gestion d'erreur. Ne les copiez pas tels-quels dans une application de production.

23.1. Programmation par événements

Les méthodes de gestion des modules Yoctopuce qui vous ont été présentées dans les chapitres précédents sont des fonctions de polling, qui consistent à demander en permanence à l'API si quelque chose a changé. Facile à appréhender, cette technique de programmation est n'est pas la plus efficace ni la plus réactive. C'est pourquoi l'API de programmation Yoctopuce propose aussi un modèle de programmation par événements. Cette technique consiste à demander à l'API de signaler elle-même les changements importants dès qu'ils sont détectés. A chaque fois qu'un paramètre clé change, l'API appelle une fonction de callback que vous avez prédéfinie.

Détecter l'arrivée et le départ des modules

La gestion du *hot-plug* est importante lorsque l'on travaille avec des modules USB, car tôt ou tard vous serez amené à brancher et débrancher un module après le lancement de votre programme. L'API a été conçue pour gérer l'arrivée et le départ inopinés des modules de manière transparente, mais votre application doit en général en tenir compte si elle veut éviter de prétendre utiliser un module qui a été débranché.

La programmation par événements est particulièrement utile pour détecter les branchements/ débranchements de modules. Il est en effet plus simple de se faire signaler les branchements, que de devoir lister en permanence les modules branchés pour en déduire ceux qui sont arrivés et ceux qui sont partis. Pour pouvoir être prévenu dès qu'un module arrive, vous avez besoin de trois morceaux de code.

Le callback
Le callback est la fonction qui sera appelée à chaque fois qu'un nouveau module Yoctopuce sera branché. Elle prend en paramètre le module concerné.

```
static void deviceArrival(YModule m)
```

```
 {
     Console.WriteLine("Nouveau module  : " + m.get_serialNumber());
 }
```

L'initialisation
Vous devez ensuite signaler à l'API qu'il faut appeler votre callback quand un nouveau module est branché.

```
  YAPI.RegisterDeviceArrivalCallback(deviceArrival);
```

Notez que si des modules sont déjà branchés lorsque le callback est enregistré, le callback sera appelé pour chacun de ces modules déjà branchés.

Déclenchement des callbacks
Un problème classique de la programmation par callbacks est que ces callbacks peuvent être appelés n'importe quand, y compris à des moments où le programme principal n'est pas prêt à les recevoir, ce qui peut avoir des effets de bords indésirables comme des dead-locks et autres conditions de course. C'est pourquoi dans l'API Yoctopuce, les callbacks d'arrivée/départs de modules ne sont appelés que pendant l'exécution de la fonction `UpdateDeviceList()`. Il vous suffit d'appeler `UpdateDeviceList()` à intervalle régulier depuis un timer ou un thread spécifique pour controller précisément quand les appels à ces callbacks auront lieu:

```
// boucle d'attente gérant les callback
while (true)
{
    // callback d'arrivée / départ de modules
    YAPI.UpdateDeviceList(ref errmsg);
    // attente non active gérant les autres callbacks
    YAPI.Sleep(500, ref errmsg);
}
```

De manière similaire, il est possible d'avoir un callback quand un module est débranché. Vous trouverez un exemple concret démontrant toutes ces techniques dans la librairie de programmation Yoctopuce de chaque langage. L'exemple se trouve dans le répertoire *Examples/Prog-EventBased*.

Attention: dans la plupart des langages, les callbacks doivent être des procédures globales, et non pas des méthodes. Si vous souhaitez que le callback appelle une méthode d'un objet, définissez votre callback sous la forme d'une procédure globale qui ensuite appellera votre méthode.

Détecter le changement de valeur d'un senseur

L'API Yoctopuce fournit aussi un système de callback permettant d'être prévenu automatiquement de la valeur d'un senseur, soit lorsqu'il a changé de manière significative, ou soit à intervalle fixe. Le code nécessaire à cet effet est assez similaire au code utilisé pour détecter l'arrivée d'un module.

Cette technique est très utile en particulier si vous voulez détecter des changements de valeur très rapides (de l'ordre de quelques millisecondes), car elle est beaucoup plus efficace (en terme de traffic sur USB) qu'une lecture répétée de la valeur et permet donc des meilleures performances.

L'appel des callbacks
Afin de permettre un meilleur contrôle du contexte d'appel, les callbacks de changement de valeurs et les callback périodiques ne sont appelés que pendant l'exécution des fonctions `YAPI.Sleep()` et `YAPI.HandleEvents()`. Vous devez donc appeler une de ces fonctions à intervalle régulier, soit depuis un timer soit depuis un thread parallèle.

```
while (true)
{
  // boucle d'attente permettant de déclencher les callbacks
  YAPI.Sleep(500, ref errmsg);
}
```

Dans les environnements de programmation où seul le thread d'interface a le droit d'interagir avec l'utilisateur, il est souvent approprié d'appeler `YAPI.HandleEvents()` depuis ce thread.

Le callback de changement de valeur

Ce type de callback est appelé lorsque un capteur générique change de manière significative. Il prend en paramètre la fonction concernée et la nouvelle valeur, sous forme d'une chaîne de caractères[1].

```
static void valueChangeCallback(YGenericSensor fct, string value)
{
  Console.WriteLine(fct.get_hardwareId() + "=" + value);
}
```

Dans la plupart des langages, les callbacks doivent être des procédures globales, et non pas des méthodes. Si vous souhaitez que le callback appelle une méthode d'un objet, définissez votre callback sous la forme d'une procédure globale qui ensuite appellera votre méthode. Si vous avez besoin de garder la référence sur votre objet, vous pouvez la stocker directement dans l'objet YGenericSensor à l'aide de la fonction `set_userData`. Il vous sera ainsi possible de la récupérer dans la procédure globale de callback en appelant `get_userData`.

Mise en place du callback de changement de valeur

Le callback est mis en place pour une fonction GenericSensor donnée à l'aide de la méthode `registerValueCallback`. L'exemple suivant met en place un callback pour la première fonction GenericSensor disponible.

```
YGenericSensor f = YGenericSensor.FirstGenericSensor();
f.registerValueCallback(valueChangeCallback);
```

Vous remarquerez que chaque fonction d'un module peut ainsi avoir un callback différent. Par ailleurs, si vous prenez goût aux callback de changement de valeur, sachez qu'il ne sont pas limités aux senseurs, et que vous pouvez les utiliser avec tous les modules Yoctopuce (par exemple pour être notifié en cas de commutation d'un relais).

Le callback périodique

Ce type de callback est automatiquement appelé à intervalle réguliers. La fréquence d'appel peut être configurée individuellement pour chaque senseur, avec des fréquences pouvant aller de cent fois par seconde à une fois par heure. Le callback prend en paramètre la fonction concernée et la valeur mesurée, sous forme d'un objet `YMeasure`. Contrairement au callback de changement de valeur qui ne contient que la nouvelle valeur instantanée, l'objet `YMeasure` peut donner la valeur minimale, moyenne et maximale observée depuis le dernier appel du callback périodique. De plus, il contient aussi l'indication de l'heure exacte qui correspond à la mesure, de sorte à pouvoir l'interpréter correctement même en différé.

```
static void periodicCallback(YGenericSensor fct, YMeasure measure)
{
  Console.WriteLine(fct.get_hardwareId() + "=" +
                    measure.get_averageValue());
}
```

Mise en place du callback périodique

Le callback est mis en place pour une fonction GenericSensor donnée à l'aide de la méthode `registerTimedReportCallback`. Pour que le callback périodique soit appelé, il faut aussi spécifier la fréquence d'appel à l'aide de la méthode `set_reportFrequency` (sinon le callback périodique est désactivé par défaut). La fréquence est spécifié sous forme textuelle (comme pour l'enregistreur de données), en spécifiant le nombre d'occurrences par seconde (/s), par minute (/m) ou par heure (/h). La fréquence maximale est 100 fois par seconde (i.e. "100/s"), et fréquence minimale est 1 fois par heure (i.e. "1/h"). Lorsque la fréquence supérieure ou égale à 1/s, la mesure représente une valeur instantanée. Lorsque la fréquence est inférieure, la mesure comporte des valeurs minimale, moyenne et maximale distinctes sur la base d'un échantillonnage effectué automatiquement par le module.

[1] La valeur passée en paramètre est la même que celle rendue par la méthode `get_advertisedValue()`

L'exemple suivant met en place un callback périodique 4 fois par minute pour la première fonction GenericSensor disponible.

```
YGenericSensor f = YGenericSensor.FirstGenericSensor();
f.set_reportFrequency("4/m");
f.registerTimedReportCallback(periodicCallback);
```

Comme pour les callback de changement valeur, chaque fonction d'un module peut avoir un callback périodique différent.

Fonction callback générique
Parfois, il est souhaitable d'utiliser la même fonction de callback pour différents types de senseurs (par exemple pour une application de mesure générique). Ceci est possible en définissant le callback pour un objet de classe `YSensor` plutôt que `YGenericSensor`. Ainsi, la même fonction callback pourra être utilisée avec toutes les sous-classes de `YSensor` (et en particulier avec `YGenericSensor`). A l'intérieur du callback, on peut utiliser la méthode `get_unit()` pour obtenir l'unité physique du capteur si nécessaire pour l'affichage.

Un exemple concret
Vous trouverez un exemple concret démontrant toutes ces techniques dans la librairie de programmation Yoctopuce de chaque langage. L'exemple se trouve dans le répertoire *Examples/Prog-EventBased*.

23.2. L'enregistreur de données

Votre Yocto-SDI12 est équipé d'un enregistreur de données, aussi appelé datalogger, capable d'enregistrer en continu les mesures effectuées par le module. La fréquence d'enregistrement maximale est de cent fois par secondes (i.e. "100/s"), et la fréquence minimale est de une fois par heure (i.e. "1/h"). Lorsque la fréquence supérieure ou égale à 1/s, la mesure représente une valeur instantanée. Lorsque la fréquence est inférieure, l'enregistreur stocke non seulement une valeur moyenne, mais aussi les valeurs minimale et maximale observées durant la période, sur la base d'un échantillonnage effectué par le module.

Notez qu'il est inutile et contre-productive de programmer une fréquence d'enregistrement plus élevée que la fréquence native d'échantillonnage du capteur concerné.

La mémoire flash de l'enregistreur de données permet d'enregistrer environ 500'000 mesures instantanées, ou 125'000 mesures moyennées. Lorsque la mémoire du datalogger est saturée, les mesures les plus anciennes sont automatiquement effacées.

Prenez garde à ne pas laisser le datalogger fonctionner inutilement à haute vitesse: le nombre d'effacements possibles d'une mémoire flash est limité (typiquement 100'000 cycles d'écriture/effacement). A la vitesse maximale, l'enregistreur peut consommer plus de 100 cycles par jour ! Notez aussi qu'il se sert à rien d'enregistrer des valeurs plus rapidement que la fréquence de mesure du capteur lui-même.

Démarrage/arrêt du datalogger

Le datalogger peut être démarré à l'aide de la méthode `set_recording()`.

```
YDataLogger l = YDataLogger.FirstDataLogger();
l.set_recording(YDataLogger.RECORDING_ON);
```

Il est possible de faire démarrer automatiquement l'enregistrement des données dès la mise sous tension du module.

```
YDataLogger l = YDataLogger.FirstDataLogger();
l.set_autoStart(YDataLogger.AUTOSTART_ON);
l.get_module().saveToFlash();  // il faut sauver le réglage!
```

Remarque: les modules Yoctopuce n'ont pas besoin d'une connection USB active pour fonctionner: ils commencent à fonctionner dès qu'ils sont alimentés. Le Yocto-SDI12 peut enregistrer des données sans être forcément raccordé à un ordinateur: il suffit d'activer le démarrage automatique du datalogger et d'alimenter le module avec un simple chargeur USB.

Effacement de la mémoire

La mémoire du datalogger peut être effacée à l'aide de la fonction `forgetAllDataStreams()`. Attention l'effacement est irréversible.

```
YDataLogger logger = YDataLogger.FirstDataLogger();
logger.forgetAllDataStreams();
```

Choix de la fréquence d'enregistrement

La fréquence d'enregistrement se configure individuellement pour chaque capteur, à l'aide de la méthode `set_logFrequency()`. La fréquence est spécifié sous forme textuelle (comme pour les callback périodiques), en spécifiant le nombre d'occurrences par seconde (/s), par minute (/m) ou par heure (/h). La valeur par défaut est "1/s".

L'exemple suivant configure la fréquence d'enregistrement à 15 mesures par minute pour le premier capteur trouvé, quel que soit son type:

```
YSensor sensor = YSensor.FirstSensor();
sensor.set_logFrequency("15/m");
```

Pour économiser la mémoire flash, il est possible de désactiver l'enregistrement des mesures pour une fonction donnée. Pour ce faire, il suffit d'utiliser la valeur "OFF":

```
sensor.set_logFrequency("OFF");
```

Limitation: Le Yocto-SDI12 ne peut pas utiliser des fréquences différentes pour les notifications périodiques et pour l'enregistrement dans le datalogger. Il est possible de désactiver l'une ou l'autre de ces fonctionnalités indépendamment, mais si les deux sont activées, elles fonctionnent nécessairement à la même fréquence.

Récupération des données

Pour récupérer les données enregistrées dans la mémoire flash du Yocto-SDI12, il faut appeler la méthode `get_recordedData()` de la fonction désirée, en spécifiant l'intervalle de temps qui nous intéresse. L'intervalle de temps est donnée à l'aide du timestamp UNIX de début et de fin. Il est aussi possible de spécifier 0 pour ne pas donner de limite de début ou de fin.

La fonction `get_recordedData()` ne retourne pas directement un tableau de valeurs mesurées, car selon la quantité de données, leur chargement pourrait potentiellement prendre trop de temps et entraver la réactivité de l'application. A la place, cette fonction retourne un objet `YDataSet` qui permet d'obtenir immédiatement une vue d'ensemble des données (résumé), puis d'en charger progressivement le détail lorsque c'est souhaitable.

Voici les principales méthodes pour accéder aux données enregistrées:

1. **dataset = sensor.get_recordedData(0,0)**: on choisit l'intervalle de temps désiré
2. **dataset.loadMore()**: pour charger les données progressivement
3. **dataset.get_summary()**: retourne une mesure unique résumant tout l'intervalle de temps
4. **dataset.get_preview()**: retourne un tableau de mesures représentant une version condensée de l'ensemble des mesures sur l'intervalle de temps choisi (réduction d'un facteur 200 environ)
5. **dataset.get_measures()**: retourne un tableau contenant toutes les mesures de l'intervalle choisi (grandit au fur et à mesure de leur chargement avec `loadMore`)

Les mesures sont des objets `YMeasure` [2]. On peut en y lire la valeur minimale, moyenne et maximale à l'aide des méthodes **get_minValue()**, **get_averageValue()** et **get_maxValue()** respectivement. Voici un petit exemple qui illustre ces fonctions:

```
// On veut récupérer toutes les données du datalogger
YDataSet dataset = sensor.get_recordedData(0, 0);

// Le 1er appel à loadMore() charge le résumé des données
dataset.loadMore();
YMeasure summary = dataset.get_summary();
string timeFmt = "dd MMM yyyy hh:mm:ss,fff";
string logFmt = "from {0} to {1} : average={2:0.00}{3}";
Console.WriteLine(String.Format(logFmt,
    summary.get_startTimeUTC_asDateTime().ToString(timeFmt),
    summary.get_endTimeUTC_asDateTime().ToString(timeFmt),
    summary.get_averageValue(), sensor.get_unit()));

// Les appels suivants à loadMore() chargent les mesures
Console.WriteLine("loading details");
int progress;
do {
    Console.Write(".");
    progress = dataset.loadMore();
} while(progress < 100);

// Ca y est, toutes les mesures sont là
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()));
}
```

Vous trouverez un exemple complet démontrant cette séquence dans la librairie de programmation Yoctopuce de chaque langage. L'exemple se trouve dans le répertoire *Examples/Prog-DataLogger*.

Horodatage

Le Yocto-SDI12 n'ayant pas de batterie, il n'est pas capable de deviner tout seul l'heure qu'il est au moment de sa mise sous tension. Néanmoins, le Yocto-SDI12 va automatiquement essayer de se mettre à l'heure de l'hôte auquel il est connecté afin de pouvoir correctement dater les mesures du datalogger:

- Lorsque le Yocto-SDI12 est branché à un ordinateur exécutant soit le VirtualHub, soit un programme quelconque utilisant la librairie Yoctopuce, il recevra l'heure de cet ordinateur.
- Lorsque le Yocto-SDI12 est branché à un YoctoHub-Ethernet, il recevra par ce biais l'heure que le YoctoHub a obtenu par le réseau (depuis un serveur du `pool.ntp.org`)
- Lorsque le Yocto-SDI12 est branché à un YoctoHub-Wireless, il recevra de celui-ci l'heure maintenue par son horloge RTC, précédemment obtenue par le réseau ou par un ordinateur.
- Lorsque le Yocto-SDI12 est branché à un appareil mobile Android, il recevra de celui-ci l'heure actuelle pour autant qu'une application utilisant la librairie Yoctopuce soit lancée.

Si aucune de ces conditions n'est remplie (par exemple si le module est simplement connecté à un chargeur USB), le Yocto-SDI12 fera de son mieux pour donner une date vraisemblable aux mesures, en repartant de l'heure des dernières mesures enregistrées. Ainsi, vous pouvez "mettre à l'heure" un Yocto-SDI12 "autonome" en le branchant sur un téléphone Android, lançant un enregistrement de données puis en le re-branchant tout seul sur un chargeur USB. Soyez toutefois conscients que, sans source de temps externe, l'horloge du Yocto-SDI12 peut dériver petit à petit (en principe jusqu'à 2%).

[2] L'objet `YMeasure` du datalogger est exactement du même type que ceux qui sont passés aux fonctions de callback périodique.

23.3. Calibration des senseurs

Votre module Yocto-SDI12 est équipé d'un capteur numérique calibré en usine. Les valeurs qu'il renvoie sont censées être raisonnablement justes dans la majorité des cas. Il existe cependant des situations où des conditions extérieures peuvent avoir une influence sur les mesures.

L'API Yoctopuce offre le moyen de re-calibrer les valeurs mesurées par votre Yocto-SDI12. Il ne n'agit pas de modifier les réglages hardware du module, mais plutôt d'effectuer une transformation a posteriori des mesures effectuées par le capteur. Cette transformation est pilotée par des paramètres qui seront stockés dans la mémoire flash du module, la rendant ainsi spécifique à chaque module. Cette re-calibration est donc entièrement software et reste parfaitement réversible.

Avant de décider de vous lancer dans la re-calibration de votre module Yocto-SDI12, assurez vous d'avoir bien compris les phénomènes qui influent sur les mesures de votre module, et que la différence en les valeurs vraies et les valeurs lues ne résultent pas d'une mauvaise utilisation ou d'un positionnement inadéquat.

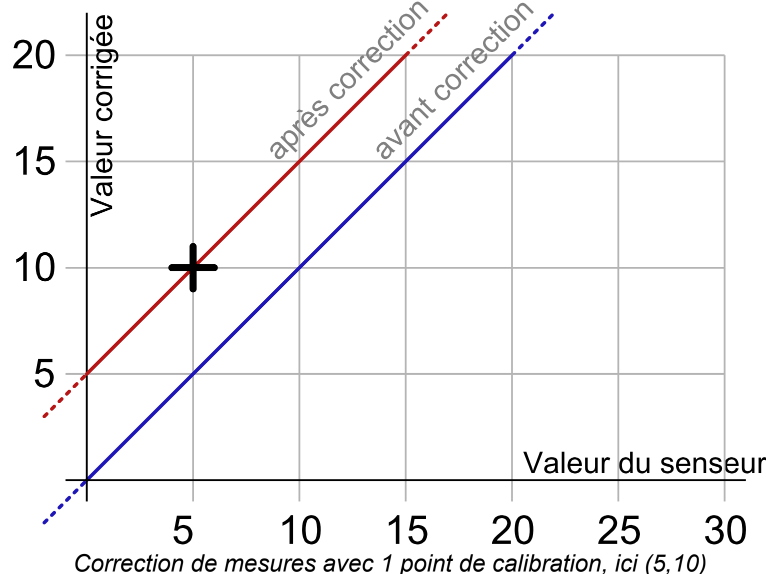
Les modules Yoctopuce supportent deux types de calibration. D'une part une interpolation linéaire basée sur 1 à 5 points de référence, qui peut être effectuée directement à l'intérieur du Yocto-SDI12. D'autre part l'API supporte une calibration arbitraire externe, implémentée à l'aide de callbacks.

Interpolation linéaire 1 à 5 points

Ces transformations sont effectuées directement dans le Yocto-SDI12 ce qui signifie que vous n'avez qu'à enregistrer les points de calibration dans la mémoire flash du module, et tous les calculs de correction seront effectués de manière totalement transparente: La fonction `get_currentValue()` renverra la valeur corrigée, alors que la fonction `get_currentRawValue()` continuera de renvoyer la valeur avant correction.

Les points de calibration sont simplement des couples *(Valeur_lue, Valeur_corrigée)*. Voyons l'influence du nombre de points de corrections sur les corrections.

Correction 1 point
La correction par 1 point ne fait qu'ajouter un décalage aux mesures. Par exemple, si vous fournissez le point de calibration *(a,b)*, toutes les valeurs mesurées seront corrigées en leur ajoutant *b-a*, de sorte à ce que quand la valeur lue sur le capteur est *a*, la fonction genericSensor1 retournera *b*.



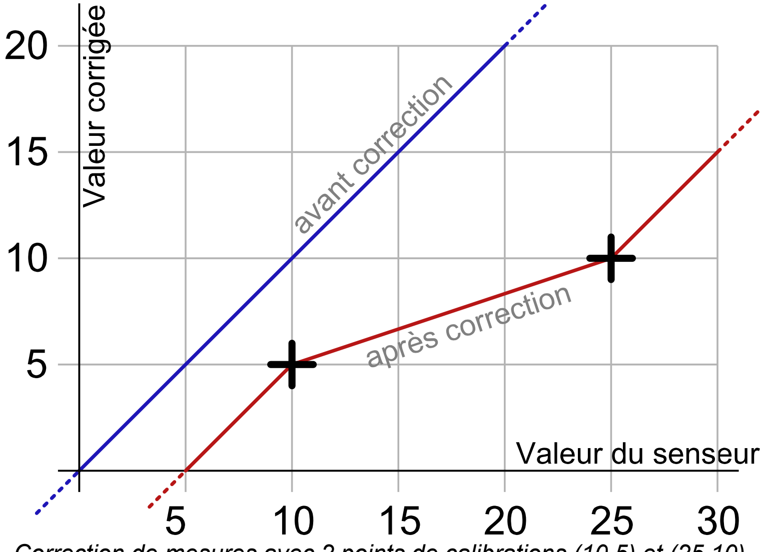
Correction de mesures avec 1 point de calibration, ici (5,10)

La mise en pratique est des plus simples: il suffit d'appeler la méthode *calibrateFromPoints()* de la fonction que l'on désire corriger. Le code suivant applique la correction illustrée sur le graphique ci-dessus à la première fonction genericSensor1 trouvée. Notez l'appel à la méthode *saveToFlash* du

module hébergeant la fonction, de manière à ce que le module n'oublie pas la calibration dès qu'il sera débranché.

```
Double[] ValuesBefore = {5};
Double[] ValuesAfter  = {10};
YGenericSensor f = YGenericSensor.FirstGenericSensor();
f.calibrateFromPoints(ValuesBefore, ValuesAfter);
f.get_module().saveToFlash();
```

Correction 2 points
La correction 2 points permet d'effectuer à la fois un décalage et une multiplication par un facteur donné entre deux points. Si vous fournissez les deux points (a,b) et (c,d), le résultat de la fonction sera multiplié par *(d-b)/(c-a)* dans l'intervalle [a,c] et décalé, de sorte à ce que quand la valeur lue par le senseur est *a* ou *c*, la fonction genericSensor1 retournera *b* ou respectivement *d*. A l'extérieur de l'intervalle [a,c], les valeurs seront simplement décalées de sorte à préserver la continuité des mesures: une augmentation de 1 sur la valeur lue par le senseur induira une augmentation de 1 sur la valeur retournée.



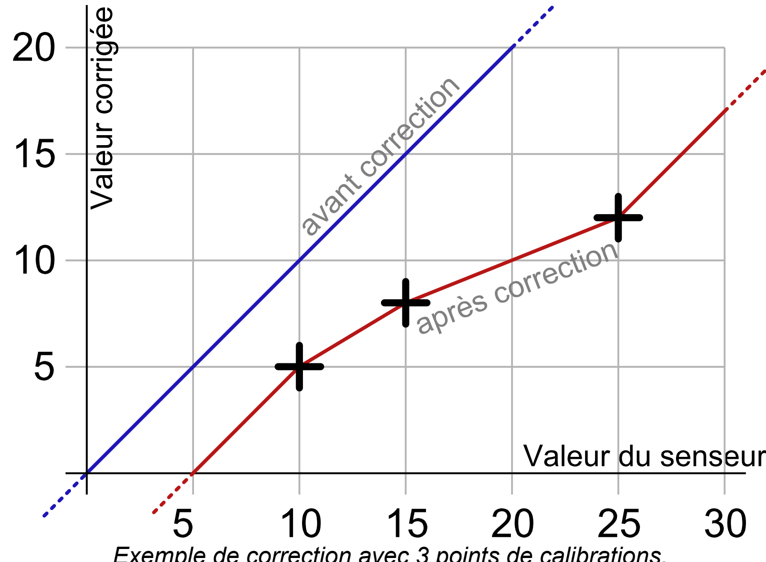
Correction de mesures avec 2 points de calibrations (10,5) et (25,10).

Le code permettant de programmer cette calibration est très similaire au code précédent

```
Double[] ValuesBefore = {10,25};
Double[] ValuesAfter  = {5,10};
YGenericSensor f = YGenericSensor.FirstGenericSensor();
f.calibrateFromPoints(ValuesBefore, ValuesAfter);
f.get_module().saveToFlash();
```

Notez que les valeurs avant correction doivent être triées dans un ordre strictement croissant, sinon elles seront purement et simplement ignorées.

Correction de 3 à 5 points
Les corrections de 3 à 5 points ne sont qu'une généralisation de la méthode à deux points, permettant de ainsi de créer jusqu' 4 intervalles de correction pour plus de précision. Ces intervalles ne peuvent pas être disjoints.



Exemple de correction avec 3 points de calibrations.

Retour à la normale
Pour annuler les effets d'une calibration sur une fonction, il suffit d'appeler la méthode *calibrateFromPoints()* avec deux tableaux vides

```
Double[] ValuesBefore = {};
Double[] ValuesAfter  = {};
YGenericSensor f = YGenericSensor.FirstGenericSensor();
f.calibrateFromPoints(ValuesBefore, ValuesAfter);
f.get_module().saveToFlash();
```

Vous trouverez dans le répertoire *Examples\Prog-Calibration* des librairies Delphi, VB et C# une application permettant d'expérimenter les effets de la calibration 1 à 5 points.

Limitations

En raison des limitations de stockage et de traitement des valeurs flottantes dans le module Yoctopuce, les valeurs des valeurs lues et des valeur corrigées doivent respecter certaines contraintes numériques:

- Seules 3 décimales sont prises en compte (résolution de 0.001)
- La valeur minimale permise est -2'100'000
- La valeur maximale permise est +2'100'000

Interpolation arbitraire

Il est aussi possible de calculer l'interpolation à la place du module, pour calculer une interpolation par spline par exemple. Il suffit pour cela d'enregistrer un callback dans l'API. Ce callback devra préciser le nombre de points de correction auquel il s'attend.

```
public static double CustomInterpolation3Points(double rawValue, int calibType,
                  int[] parameters, double[] beforeValues, double[] afterValues)
  {  double result;
     // la valeur a corriger est rawValue
     // les points de calibrations sont dans beforeValues et afterValues
     result = ....    // interpolation de votre choix
     return result;
   }
YAPI.RegisterCalibrationHandler(3, CustomInterpolation3Points);
```

Notez que ces callbacks d'interpolation sont globaux, et non pas spécifiques à chaque fonction. Ainsi à chaque fois que quelqu'un demandera une valeur à un module qui disposera dans sa mémoire flash du bon nombre de points de calibration, le callback correspondant sera appelé pour corriger la

valeur avant de la renvoyer, permettant ainsi de corriger les mesures de manière totalement transparente.

24. Mise à jour du firmware

Il existe plusieurs moyens de mettre à jour le firmware des modules Yoctopuce.

24.1. Le VirtualHub ou le YoctoHub

Il est possible de mettre à jour un module directement depuis l'interface web du VirutalHub ou du YoctoHub. Il suffit d'accéder à la fenêtre de configuration du module que à mettre à jour et de cliquer sur le bouton "upgrade". Le VirtualHub démarre un assistant qui vous guidera durant la procédure de mise à jour.

Si pour une raison ou une autre, la mise à jour venait à échouer et que le module de fonctionnait plus, débranchez puis rebranchez le module en maintenant sur le Yocto-bouton appuyé. Le module va démarrer en mode "mise à jour" et sera listé en dessous des modules connectés.

24.2. La librairie ligne de commandes

Tous les outils en lignes de commandes ont la possibilité de mettre à jour les modules Yoctopuce grâce à la commande `downloadAndUpdate`. Le mécanisme de sélection des modules fonctionne comme pour une commande traditionnelle. La [cible] est le nom du module qui va être mis à jour. Vous pouvez aussi utiliser les alias "any" ou "all", ou encore une liste de noms, séparés par des virgules, sans espace.

```
C:\>Executable [options] [cible] commande [paramètres]
```

L'exemple suivant met à jour tous les modules Yoctopuce connectés en 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:\>
```

24.3. L'application Android Yocto-Firmware

Il est possible de mettre à jour le firmware de vos modules depuis votre téléphone ou tablette Android avec l'application Yocto-Firmware. Cette application liste tous les modules Yoctopuce

branchés en USB et vérifie si un firmware plus récent est disponible sur www.yoctopuce.com. Si un firmware plus récent est disponible, il est possible de mettre à jour le module. L'application se charge de télécharger et d'installer le nouveau firmware en préservant les paramètres du module.

Attention, pendant la mise à jour du firmware, le module redémarre plusieurs fois. Android interprète le reboot d'un périphérique USB comme une déconnexion et reconnexion du périphérique USB, et demande à nouveau l'autorisation d'utiliser le port USB. L'utilisateur est obligé de cliquer sur **OK** pour que la procédure de mise à jour se termine correctement.

24.4. La librairie de programmation

Si vous avez besoin d'intégrer la mise à jour de firmware dans votre application, les librairies proposent une API pour mettre à jour vos modules.

Sauvegarder et restaurer les paramètres

La méthode `get_allSettings()` retourne un buffer binaire qui permet de sauvegarder les paramètres persistants d'un module. Cette fonction est très utile pour sauvegarder la configuration réseau d'un YoctoHub par exemple.

```
YWireless wireless = YWireless.FindWireless("reference");
YModule m = wireless.get_module();
byte[] default_config =  m.get_allSettings();
saveFile("default.bin", default_config);
...
```

Ces paramètres peuvent être appliqués sur d'autres modules à l'aide de la méthode `set_allSettings()`.

```
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();
}
```

Chercher le bon firmware

La première étape pour mettre à jour un module Yoctopuce est de trouver quel firmware il faut utiliser, c'est le travail de la méthode `checkFirmware(path, onlynew)` de l'objet YModule. Cette méthode vérifie que le firmware passé en argument (`path`) est compatible avec le module. Si le paramètre `onlynew` est vrai, cette méthode vérifie si le firmware est plus récent que la version qui est actuellement utilisée par le module. Quand le fichier n'est pas compatible (ou si le fichier est plus vieux que la version installée), cette méthode retourne une chaîne vide. Si au contraire le fichier est valide, la méthode retourne le chemin d'accès d'un fichier.

Le code suivant vérifie si le fichier `c:\tmp\METEOMK1.17328.byn` est compatible avec le module stocké dans la variable *m*.

```
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");
}
...
```

Il est possible de passer un répertoire en argument (au lieu d'un fichier). Dans ce cas la méthode va parcourir récursivement tous les fichiers du répertoire et retourner le firmware compatible le plus récent. Le code suivant vérifie s'il existe un firmware plus récent dans le répertoire `c:\tmp\`.

```
YModule m = YModule.FirstModule();
...
...
string path = "c:\\tmp";
string newfirm = m.checkFirmware(path, true);
if (newfirm != "") {
  Console.WriteLine("firmware " + newfirm + " is compatible and newer");
}
...
```

Il est aussi possible de passer la chaîne "www.yoctopuce.com" en argument pour vérifier s'il existe un firmware plus récent publié sur le site web de Yoctopuce. Dans ce cas, la méthode retournera l'URL du firmware. Vous pourrez soit utiliser cette URL pour télécharger le firmware sur votre disque, soit utiliser cette URL lors de la mise à jour du firmware (voir ci-dessous). Bien évidemment, cette possibilité ne fonctionne que si votre machine est reliée à Internet.

```
YModule m = YModule.FirstModule();
...
...
string url = m.checkFirmware("www.yoctopuce.com", true);
if (url != "") {
  Console.WriteLine("new firmware is available at " + url );
}
...
```

Mettre à jour le firmware

La mise à jour du firmware peut prendre plusieurs minutes, c'est pourquoi le processus de mise à jour est exécuté par la librairie en arrière plan et est contrôlé par le code utilisateur à l'aide de la classe `YFirmwareUdpate`.

Pour mettre à jour un module Yoctopuce, il faut obtenir une instance de la classe `YFirmwareUpdate` à l'aide de la méthode `updateFirmware` d'un objet YModule. Le seul paramètre de cette méthode est le *path* du firmware à installer. Cette méthode ne démarre pas immédiatement la mise à jour, mais retourne un objet `YFirmwareUpdate` configuré pour mettre à jour le module.

```
string newfirm = m.checkFirmware("www.yoctopuce.com", true);
.....
YFirmwareUpdate fw_update = m.updateFirmware(newfirm);
```

La méthode `startUpdate()` démarre la mise à jour en arrière plan. Ce processus en arrière plan se charge automatiquement de:

1. sauvegarder des paramètres du module,
2. redémarrer le module en mode "mise à jour"
3. mettre à jour le firmware
4. démarrer le module avec la nouvelle version du firmware
5. restaurer les paramètres

Les méthodes `get_progress()` et `get_progressMessage()` permettent de suivre la progression de la mise à jour. `get_progress()` retourne la progression sous forme de pourcentage (100 = mise à jour terminée). `get_progressMessage()` retourne une chaîne de caractères décrivant l'opération en cours (effacement, écriture, reboot,...). Si la méthode `get_progress()` retourne une valeur négative, c'est que le processus de mise à jour à échoué. Dans ce cas la méthode `get_progressMessage()` retourne le message d'erreur.

Le code suivant démarre la mise à jour et affiche la progression sur la sortie standard.

```
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!");
  }
```

Particularité d'Android

Il est possible de mettre à jour un firmware d'un module en utilisant la librairie Android. Mais pour les modules branchés en USB, Android va demander à l'utilisateur d'autoriser l'application à accéder au port USB.

Pendant la mise à jour du firmware, le module redémarre plusieurs fois. Android interprète le reboot d'un périphérique USB comme une déconnexion et reconnexion du port USB, et interdit tout accès USB tant que l'utilisateur n'a pas fermé le pop-up. L'utilisateur est obligé de cliquer sur *OK* pour que la procédure de mise à jour puisse continuer correctement. **Il n'est pas possible de mettre à jour un module branché en USB à un appareil Android sans que l'utilisateur ne soit obligé d'interagir avec l'appareil**.

24.5. Le mode "mise à jour"

Si vous désirez effacer tous les paramètres du module ou que votre module ne démarre plus correctement, il est possible d'installer un firmware depuis le mode "mise à jour".

Pour forcer le module à fonctionner dans le mode "mis à jour", débranchez-le, attendez quelques secondes, et rebranchez-le en maintenant le *Yocto-Bouton* appuyé. Cela a pour effet de faire démarrer le module en mode "mise à jour". Ce mode de fonctionnement est protégé contre les corruptions et est toujours accessible.

Dans ce mode, le module n'est plus détecté par les objets YModules. Pour obtenir la liste des modules connectés en mode "mise à jour", il faut utiliser la fonction `YAPI.GetAllBootLoaders()`. Cette fonction retourne un tableau de chaînes de caractères avec le numéro de série des modules en le mode "mise à jour".

```
List<string> allBootLoader = YAPI.GetAllBootLoaders();
```

La procédure de mise à jour est identique au cas standard (voir section précédente), mais il faut instancier manuellement l'objet `YFirmwareUpdate` au lieu d'appeler `module.updateFirmware()`. Le constructeur prend en argument trois paramètres: le numéro de série du module, le path du firmware à installer, et un tableau de bytes avec les paramètres à restaurer à la fin de la mise à jour (ou `null` pour restaurer les paramètres d'origine).

```
YFirmwareUpdateupdate fw_update;
fw_update = new YFirmwareUpdate(allBootLoader[0], newfirm, null);
int status = fw_update.startUpdate();
.....
```

25. Référence de l'API de haut niveau

Ce chapitre résume les fonctions de l'API de haut niveau pour commander votre Yocto-SDI12. La syntaxe et les types précis peuvent varier d'un langage à l'autre mais, sauf avis contraire toutes sont disponibles dans chaque language. Pour une information plus précise sur les types des arguments et des valeurs de retour dans un langage donné, veuillez vous référer au fichier de définition pour ce langage (`yocto_api.*` ainsi que les autres fichiers `yocto_*` définissant les interfaces des fonctions).

Dans les langages qui supportent les exceptions, toutes ces fonctions vont par défaut générer des exceptions en cas d'erreur plutôt que de retourner la valeur d'erreur documentée pour chaque fonction, afin de faciliter le déboguage. Il est toutefois possible de désactiver l'utilisation d'exceptions à l'aide de la fonction `yDisableExceptions()`, si l'on préfère travailler avec des valeurs de retour d'erreur.

Ce chapitre ne reprend pas en détail les concepts de programmation décrits plus tôt, afin d'offrir une référence plus concise. En cas de doute, n'hésitez pas à retourner au chapitre décrivant en détail de chaque attribut configurable.

25.1. La classe YAPI

Fonctions générales

Ces quelques fonctions générales permettent l'initialisation et la configuration de la librairie Yoctopuce. Dans la plupart des cas, un appel à `yRegisterHub()` suffira en tout et pour tout. Ensuite, vous pourrez appeler la fonction globale `yFind...()` ou `yFirst...()` correspondant à votre module pour pouvoir interagir avec lui.

Pour utiliser les fonctions décrites ici, vous devez inclure:

java	import com.yoctopuce.YoctoAPI.YAPI;
dnp	import YoctoProxyAPI.YAPIProxy
cp	#include "yocto_api_proxy.h"
ml	import YoctoProxyAPI.YAPIProxy"
js	<script type='text/javascript' src='yocto_api.js'></script>
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
uwp	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *
php	require_once('yocto_api.php');
ts	in HTML: import { YAPI, YErrorMsg, YModule, YSensor } from '../../dist/esm/yocto_api_browser.js'; in Node.js: import { YAPI, YErrorMsg, YModule, YSensor } from 'yoctolib-cjs/yocto_api_nodejs.js';
es	in HTML: <script src="../../lib/yocto_api.js"></script> in node.js: require('yoctolib-es2017/yocto_api.js');
vi	YModule.vi

Fonction globales

YAPI.AddUdevRule(force)
Ajoute une règle UDEV qui autorise tous les utilisateurs à accéder aux modules Yoctopuce connectés sur les ports USB.
cpp m pas vb cs java uwp py php ts es

YAPI.CheckLogicalName(name)
Vérifie si un nom donné est valide comme nom logique pour un module ou une fonction.
cpp m pas vb cs java uwp py php ts es

YAPI.ClearHTTPCallbackCacheDir(removeFiles)
Désactive le cache de callback HTTP.
java php

YAPI.DisableExceptions()
Désactive l'utilisation d'exceptions pour la gestion des erreurs.
cpp m pas vb cs uwp py php ts es

YAPI.EnableExceptions()
Réactive l'utilisation d'exceptions pour la gestion des erreurs.
cpp m pas vb cs uwp py php ts es

YAPI.EnableUSBHost(osContext)

Cette fonction est utilisée uniquement sous Android.

java

YAPI.FreeAPI()

Attends que toutes les communications en cours avec les modules Yoctopuce soient terminées puis libère les ressources utilisées par la librairie Yoctopuce.

cpp m pas vb cs java uwp py php ts es dnp

YAPI.GetAPIVersion()

Retourne la version de la librairie Yoctopuce utilisée.

cpp m pas vb cs java uwp py php ts es dnp

YAPI.GetCacheValidity()

Retourne la durée de validité des données chargée par la libraire.

cpp m pas vb cs java uwp py php ts es

YAPI.GetDeviceListValidity()

Retourne le délai entre chaque énumération forcée des YoctoHubs utilisés.

cpp m pas vb cs java uwp py php ts es

YAPI.GetDllArchitecture()

Retourne l'architecture du binaire de la librairie de communication Yoctopuce utilisée.

dnp

YAPI.GetDllPath()

Retourne l'emplacement sur le disque des librairies Yoctopuce utilisées.

dnp

YAPI.GetLog(lastLogLine)

Récupère les messages de logs de la librairie de communication Yoctopuce.

dnp

YAPI.GetNetworkTimeout()

Retourne le délai de connexion réseau pour `yRegisterHub()` et `yUpdateDeviceList()`.

cpp m pas vb cs java uwp py php ts es dnp

YAPI.GetTickCount()

Retourne la valeur du compteur monotone de temps (en millisecondes).

cpp m pas vb cs java uwp py php ts es

YAPI.HandleEvents(errmsg)

Maintient la communication de la librairie avec les modules Yoctopuce.

cpp m pas vb cs java uwp py php ts es

YAPI.InitAPI(mode, errmsg)

Initialise la librairie de programmation de Yoctopuce explicitement.

cpp m pas vb cs java uwp py php ts es

YAPI.PreregisterHub(url, errmsg)

Alternative plus tolérante à `yRegisterHub()`.

cpp m pas vb cs java uwp py php ts es dnp

YAPI.RegisterDeviceArrivalCallback(arrivalCallback)

Enregistre une fonction de callback qui sera appelée à chaque fois qu'un module est branché.

cpp m pas vb cs java uwp py php ts es

YAPI.RegisterDeviceRemovalCallback(removalCallback)

Enregistre une fonction de callback qui sera appelée à chaque fois qu'un module est débranché.

cpp m pas vb cs java uwp py php ts es

YAPI.RegisterHub(url, errmsg)

Configure la librairie Yoctopuce pour utiliser les modules connectés sur une machine donnée.

cpp m pas vb cs java uwp py php ts es dnp

YAPI.RegisterHubDiscoveryCallback(hubDiscoveryCallback)

Enregistre une fonction de callback qui est appelée chaque fois qu'un hub réseau s'annonce avec un message SSDP.

cpp m pas vb cs java uwp py ts es

YAPI.RegisterHubWebsocketCallback(ws, errmsg, authpwd)

Variante de la fonction `yRegisterHub()` destinée à initialiser l'API Yoctopuce sur une session Websocket existante, dans le cadre d'un callback WebSocket entrant.

YAPI.RegisterLogFunction(logfun)

Enregistre une fonction de callback qui sera appellée à chaque fois que l'API a quelque chose à dire.

cpp m pas vb cs java uwp py ts es

YAPI.SelectArchitecture(arch)

Sélectionne manuellement l'architecture de la libraire dynamique à utiliser pour accéder à USB.

py

YAPI.SetCacheValidity(cacheValidityMs)

Change la durée de validité des données chargées par la librairie.

cpp m pas vb cs java uwp py php ts es

YAPI.SetDelegate(object)

(Objective-C uniquement) Enregistre un objet délégué qui doit se conformer au protocole `YDeviceHotPlug`.

m

YAPI.SetDeviceListValidity(deviceListValidity)

Change le délai entre chaque énumération forcée des YoctoHub utilisés.

cpp m pas vb cs java uwp py php ts es

YAPI.SetHTTPCallbackCacheDir(directory)

Active le cache du callback HTTP.

java php

YAPI.SetNetworkTimeout(networkMsTimeout)

Change le délai de connexion réseau pour `yRegisterHub()` et `yUpdateDeviceList()`.

cpp m pas vb cs java uwp py php ts es dnp

YAPI.SetTimeout(callback, ms_timeout, args)

Appelle le callback spécifié après un temps d'attente spécifié.

ts es

YAPI.SetUSBPacketAckMs(pktAckDelay)

Active la quittance des paquets USB reçus par la librairie Yoctopuce.

java

YAPI.Sleep(ms_duration, errmsg)

Effectue une pause dans l'exécution du programme pour une durée spécifiée.

cpp m pas vb cs java uwp py php ts es

YAPI.TestHub(url, mstimeout, errmsg)

Test si un hub est joignable.

cpp m pas vb cs java uwp py php ts es dnp

YAPI.TriggerHubDiscovery(errmsg)

Relance une détection des hubs réseau.

cpp m pas vb cs java uwp py ts es

YAPI.UnregisterHub(url)

Configure la librairie Yoctopuce pour ne plus utiliser les modules connectés sur une machine préalablement enregistrer avec RegisterHub.

cpp m pas vb cs java uwp py php ts es

YAPI.UpdateDeviceList(errmsg)

Force une mise-à-jour de la liste des modules Yoctopuce connectés.

cpp m pas vb cs java uwp py php ts es

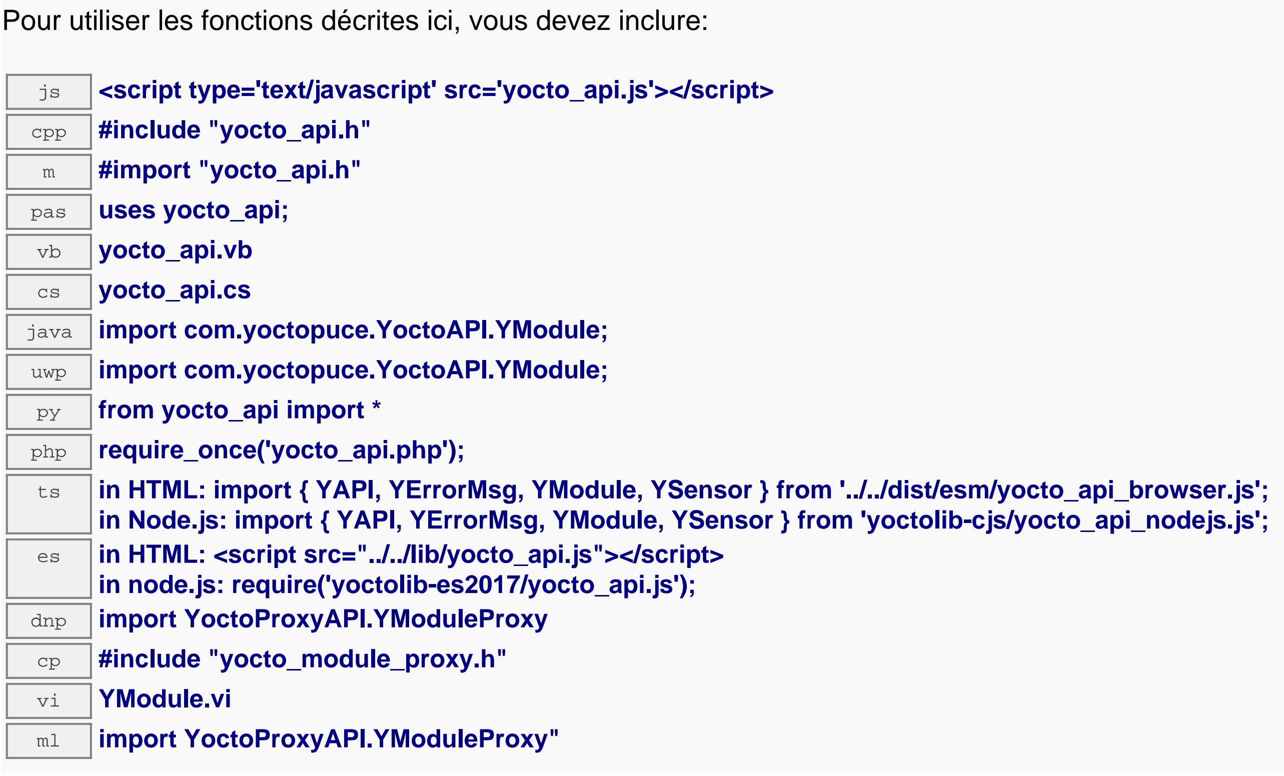
YAPI.UpdateDeviceList_async(callback, context)

Force une mise-à-jour de la liste des modules Yoctopuce connectés.

25.2. La classe YModule

Interface de contrôle des paramètres généraux des modules Yoctopuce

La classe `YModule` est utilisable avec tous les modules USB de Yoctopuce. Elle permet de contrôler les paramètres généraux du module, et d'énumérer les fonctions fournies par chaque module.



Pour utiliser les fonctions décrites ici, vous devez inclure:

js	<script type='text/javascript' src='yocto_api.js'></script>
cpp	#include "yocto_api.h"
m	#import "yocto_api.h"
pas	uses yocto_api;
vb	yocto_api.vb
cs	yocto_api.cs
java	import com.yoctopuce.YoctoAPI.YModule;
uwp	import com.yoctopuce.YoctoAPI.YModule;
py	from yocto_api import *
php	require_once('yocto_api.php');
ts	in HTML: import { YAPI, YErrorMsg, YModule, YSensor } from '../../dist/esm/yocto_api_browser.js'; in Node.js: import { YAPI, YErrorMsg, YModule, YSensor } from 'yoctolib-cjs/yocto_api_nodejs.js';
es	in HTML: <script src="../../lib/yocto_api.js"></script> in node.js: require('yoctolib-es2017/yocto_api.js');
dnp	import YoctoProxyAPI.YModuleProxy
cp	#include "yocto_module_proxy.h"
vi	YModule.vi
ml	import YoctoProxyAPI.YModuleProxy"

Fonction globales

YModule.FindModule(func)

Permet de retrouver un module d'après son numéro de série ou son nom logique.

cpp m pas vb cs java uwp py php ts es dnp

YModule.FindModuleInContext(yctx, func)

Permet de retrouver un module d'après un identifiant donné dans un Context YAPI.

java uwp ts es

YModule.FirstModule()

Commence l'énumération des modules accessibles par la librairie.

cpp m pas vb cs java uwp py php ts es

Propriétés des objets `YModuleProxy`

module→Beacon *[modifiable]*

état de la balise de localisation.

dnp

module→FirmwareRelease *[lecture seule]*

Version du logiciel embarqué du module.

dnp

module→FunctionId *[lecture seule]*

Identifiant matériel de la *n*ième fonction du module.

dnp

module→HardwareId ***[lecture seule]***

Identifiant unique du module.

dnp

module→IsOnline ***[lecture seule]***

Vérifie si le module est joignable.

dnp

module→LogicalName ***[modifiable]***

Nom logique du module.

dnp

module→Luminosity ***[modifiable]***

Luminosité des leds informatives du module (valeur entre 0 et 100).

dnp

module→ProductId ***[lecture seule]***

Identifiant USB du module, préprogrammé en usine.

dnp

module→ProductName ***[lecture seule]***

Nom commercial du module, préprogrammé en usine.

dnp

module→ProductRelease ***[lecture seule]***

Numéro uméro de révision du module hardware, préprogrammé en usine.

dnp

module→SerialNumber ***[lecture seule]***

Numéro de série du module, préprogrammé en usine.

dnp

Méthodes des objets `YModule`

module→addFileToHTTPCallback(filename)

Ajoute un fichier aux données uploadées lors du prochain callback HTTP.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→checkFirmware(path, onlynew)

Teste si le fichier byn est valide pour le module.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→clearCache()

Invalide le cache.

cpp m pas vb cs java py php ts es

module→describe()

Retourne un court texte décrivant le module.

cpp m pas vb cs java py php ts es

module→download(pathname)

Télécharge le fichier choisi du module et retourne son contenu.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→functionBaseType(functionIndex)

Retourne le type de base de la *n*ième fonction du module.

cpp pas vb cs java py php ts es

module→functionCount()

Retourne le nombre de fonctions (sans compter l'interface "module") existant sur le module.

cpp m pas vb cs java py php ts es

module→functionId(functionIndex)

Retourne l'identifiant matériel de la *n*ième fonction du module.

cpp m pas vb cs java py php ts es

module→functionName(functionIndex)

Retourne le nom logique de la *n*ième fonction du module.

cpp m pas vb cs java py php ts es

module→functionType(functionIndex)

Retourne le type de la *n*ième fonction du module.

cpp pas vb cs java py php ts es

module→functionValue(functionIndex)

Retourne la valeur publiée par la *n*ième fonction du module.

cpp m pas vb cs java py php ts es

module→get_allSettings()

Retourne tous les paramètres de configuration du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→get_beacon()

Retourne l'état de la balise de localisation.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→get_errorMessage()

Retourne le message correspondant à la dernière erreur survenue lors de l'utilisation de l'objet module.

cpp m pas vb cs java py php ts es

module→get_errorType()

Retourne le code d'erreur correspondant à la dernière erreur survenue lors de l'utilisation de l'objet module.

cpp m pas vb cs java py php ts es

module→get_firmwareRelease()

Retourne la version du logiciel embarqué du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→get_functionIds(funType)

Retourne les identifiants matériels des fonctions correspondant au type passé en argument.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→get_hardwareId()

Retourne l'identifiant unique du module.

cpp m vb cs java py php ts es dnp pas uwp cmd

module→get_icon2d()

Retourne l'icône du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→get_lastLogs()

Retourne une chaîne de charactère contenant les derniers logs du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→get_logicalName()

Retourne le nom logique du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→get_luminosity()

Retourne la luminosité des leds informatives du module (valeur entre 0 et 100).

cpp m pas vb cs java uwp py php ts es dnp cmd

module→get_parentHub()

Retourne le numéro de série du YoctoHub sur lequel est connecté le module.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→get_persistentSettings()

Retourne l'état courant des réglages persistents du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→get_productId()

Retourne l'identifiant USB du module, préprogrammé en usine.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→get_productName()

Retourne le nom commercial du module, préprogrammé en usine.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→get_productRelease()

Retourne le numéro uméro de révision du module hardware, préprogrammé en usine.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→get_rebootCountdown()

Retourne le nombre de secondes restantes avant un redémarrage du module, ou zéro si aucun redémarrage n'a été agendé.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→get_serialNumber()

Retourne le numéro de série du module, préprogrammé en usine.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→get_subDevices()

Retourne la liste des modules branchés au module courant.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→get_upTime()

Retourne le numbre de millisecondes écoulées depuis la mise sous tension du module

cpp m pas vb cs java uwp py php ts es dnp cmd

module→get_url()

Retourne l'URL utilisée pour accéder au module.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→get_usbCurrent()

Retourne le courant consommé par le module sur le bus USB, en milliampères.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→get_userData()

Retourne le contenu de l'attribut userData, précédemment stocké à l'aide de la méthode `set_userData`.

cpp m pas vb cs java py php ts es

module→get_userVar()

Retourne la valeur entière précédemment stockée dans cet attribut.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→hasFunction(funcId)

Teste la présence d'une fonction pour le module courant.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→isOnline()

Vérifie si le module est joignable, sans déclencher d'erreur.

cpp m pas vb cs java py php ts es dnp

module→isOnline_async(callback, context)

Vérifie si le module est joignable, sans déclencher d'erreur.

module→load(msValidity)

Met en cache les valeurs courantes du module, avec une durée de validité spécifiée.

cpp m pas vb cs java py php ts es

module→load_async(msValidity, callback, context)

Met en cache les valeurs courantes du module, avec une durée de validité spécifiée.

module→log(text)

Ajoute un message arbitraire dans les logs du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→nextModule()

Continue l'énumération des modules commencée à l'aide de `yFirstModule()`.

cpp m pas vb cs java uwp py php ts es

module→reboot(secBeforeReboot)

Agende un simple redémarrage du module dans un nombre donné de secondes.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→registerBeaconCallback(callback)

Enregistre une fonction de callback qui sera appelée à chaque changement d'état de la balise de localisation du module.

cpp m pas vb cs java uwp py php ts es

module→registerConfigChangeCallback(callback)

Enregistre une fonction de callback qui sera appelée à chaque fois qu'un réglage persistant d'un module est modifié (par exemple changement d'unité de mesure, etc.)

cpp m pas vb cs java uwp py php ts es

module→registerLogCallback(callback)

Enregistre une fonction de callback qui sera appelée à chaque fois le module émet un message de log.

cpp m pas vb cs java uwp py php ts es

module→revertFromFlash()

Recharge les réglages stockés dans le mémoire non volatile du module, comme à la mise sous tension du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→saveToFlash()

Sauve les réglages courants dans la mémoire non volatile du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→set_allSettings(settings)

Rétablit tous les paramètres du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→set_allSettingsAndFiles(settings)

Rétablit tous les paramètres de configuration et fichiers sur un module.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→set_beacon(newval)

Allume ou éteint la balise de localisation du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→set_logicalName(newval)

Change le nom logique du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→set_luminosity(newval)

Modifie la luminosité des leds informatives du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→set_userData(data)

Enregistre un contexte libre dans l'attribut userData de la fonction, afin de le retrouver plus tard à l'aide de la méthode `get_userData`.

cpp m pas vb cs java py php ts es

module→set_userVar(newval)

Stocke une valeur 32 bits dans la mémoire volatile du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→triggerConfigChangeCallback()

Force le déclenchement d'un callback de changement de configuration, afin de vérifier s'ils sont disponibles ou pas.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→triggerFirmwareUpdate(secBeforeReboot)

Agende un redémarrage du module en mode spécial de reprogrammation du logiciel embarqué.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→updateFirmware(path)

Prepare une mise à jour de firmware du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→updateFirmwareEx(path, force)

Prepare une mise à jour de firmware du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

module→wait_async(callback, context)

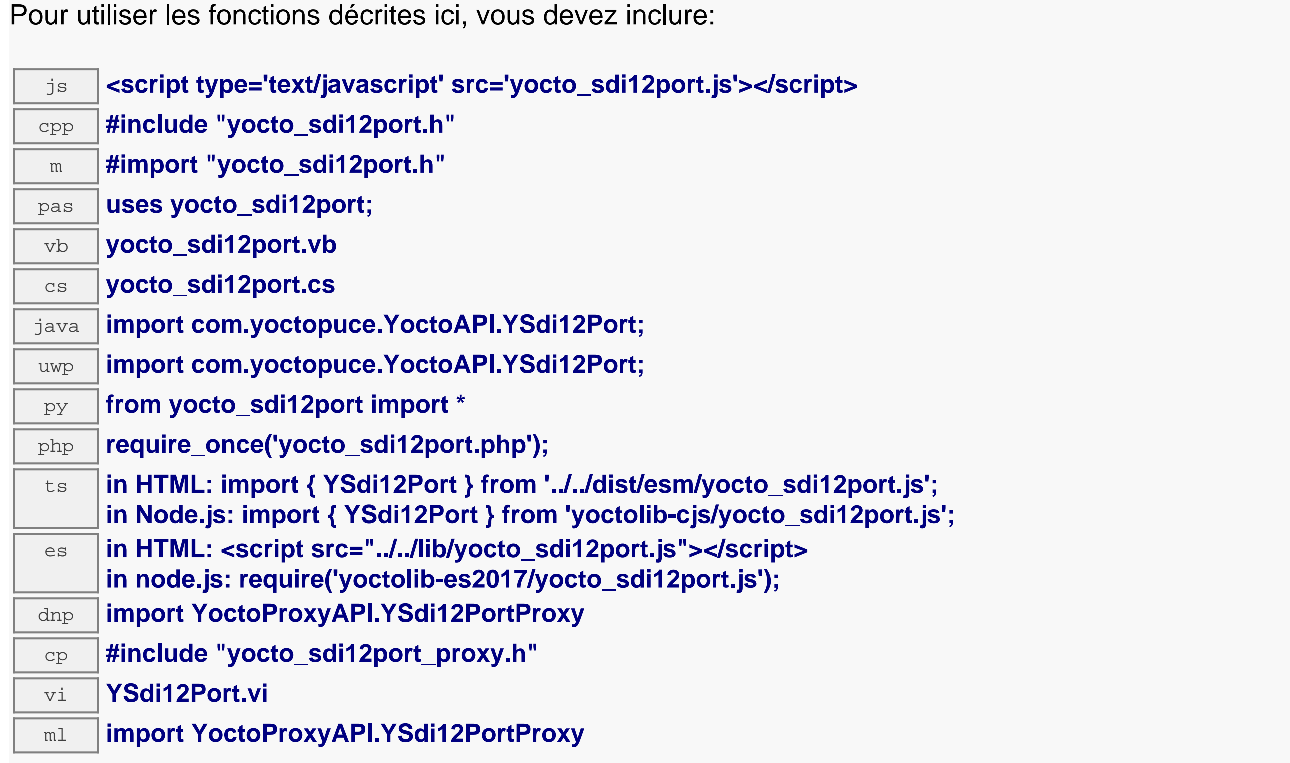
Attend que toutes les commandes asynchrones en cours d'exécution sur le module soient terminées, et appelle le callback passé en paramètre.

ts es

25.3. La classe YSdi12Port

Interface pour intéragir avec les ports SDI12

La classe `YSdi12Port` permet de piloter entièrement un module d'interface SDI12 Yoctopuce. Elle permet d'envoyer et de recevoir des données, et de configurer les paramètres de transmission (vitesse, nombre de bits, parité, contrôle de flux et protocole). Notez que les interfaces SDI12 Yoctopuce ne sont pas visibles comme des ports COM virtuels. Ils sont faits pour être utilisés comme tous les autres modules Yoctopuce.



Pour utiliser les fonctions décrites ici, vous devez inclure:

js	<script type='text/javascript' src='yocto_sdi12port.js'></script>
cpp	#include "yocto_sdi12port.h"
m	#import "yocto_sdi12port.h"
pas	uses yocto_sdi12port;
vb	yocto_sdi12port.vb
cs	yocto_sdi12port.cs
java	import com.yoctopuce.YoctoAPI.YSdi12Port;
uwp	import com.yoctopuce.YoctoAPI.YSdi12Port;
py	from yocto_sdi12port import *
php	require_once('yocto_sdi12port.php');
ts	in HTML: import { YSdi12Port } from '../../dist/esm/yocto_sdi12port.js'; in Node.js: import { YSdi12Port } from 'yoctolib-cjs/yocto_sdi12port.js';
es	in HTML: <script src="../../lib/yocto_sdi12port.js"></script> in node.js: require('yoctolib-es2017/yocto_sdi12port.js');
dnp	import YoctoProxyAPI.YSdi12PortProxy
cp	#include "yocto_sdi12port_proxy.h"
vi	YSdi12Port.vi
ml	import YoctoProxyAPI.YSdi12PortProxy

Fonction globales

YSdi12Port.FindSdi12Port(func)

Permet de retrouver un port SDI12 d'après un identifiant donné.

cpp m pas vb cs java uwp py php ts es dnp

YSdi12Port.FindSdi12PortInContext(yctx, func)

Permet de retrouver un port SDI12 d'après un identifiant donné dans un Context YAPI.

java uwp ts es

YSdi12Port.FirstSdi12Port()

Commence l'énumération des ports SDI12 accessibles par la librairie.

cpp m pas vb cs java uwp py php ts es

YSdi12Port.FirstSdi12PortInContext(yctx)

Commence l'énumération des ports SDI12 accessibles par la librairie.

java uwp ts es

YSdi12Port.GetSimilarFunctions()

Enumère toutes les fonctions de type Sdi12Port disponibles sur les modules actuellement joignables par la librairie, et retourne leurs identifiants matériels uniques (hardwareId).

dnp

Propriétés des objets `YSdi12PortProxy`

sdi12port→AdvertisedValue *[lecture seule]*

Courte chaîne de caractères représentant l'état courant de la fonction.

dnp

sdi12port→FriendlyName *[lecture seule]*

Identifiant global de la fonction au format `NOM_MODULE.NOM_FONCTION`.

dnp

sdi12port→FunctionId *[lecture seule]*

Identifiant matériel du port SDI12, sans référence au module.

dnp

sdi12port→HardwareId *[lecture seule]*

Identifiant matériel unique de la fonction au format `SERIAL.FUNCTIONID`.

dnp

sdi12port→IsOnline *[lecture seule]*

Vérifie si le module hébergeant la fonction est joignable, sans déclencher d'erreur.

dnp

sdi12port→JobMaxSize *[lecture seule]*

Taille maximale d'un fichier job.

dnp

sdi12port→JobMaxTask *[lecture seule]*

Nombre maximal de tâches dans un job supporté par le module.

dnp

sdi12port→LogicalName *[modifiable]*

Nom logique de la fonction.

dnp

sdi12port→Protocol *[modifiable]*

Type de protocole utilisé sur la communication série, sous forme d'une chaîne de caractères.

dnp

sdi12port→SerialMode *[modifiable]*

S paramètres de communication du port, sous forme d'une chaîne de caractères du type "1200,7E1,Simplex".

dnp

sdi12port→SerialNumber *[lecture seule]*

Numéro de série du module, préprogrammé en usine.

dnp

sdi12port→StartupJob *[modifiable]*

Nom du fichier de tâches à exécuter au démarrage du module.

dnp

sdi12port→VoltageLevel *[modifiable]*

Niveau de tension utilisé par le module sur le port série.

dnp

Méthodes des objets `YSdi12Port`

sdi12port→changeAddress(oldAddress, newAddress)

Change l'adresse du capteur sélectionné, et retourne les informations du capteur avec la nouvelle adresse.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→clearCache()

Invalide le cache.

cpp m pas vb cs java py php ts es

sdi12port→describe()

Retourne un court texte décrivant de manière non-ambigüe l'instance du port SDI12 au format `TYPE(NAME)=SERIAL.FUNCTIONID`.

cpp m pas vb cs java py php ts es

sdi12port→discoverAllSensors()

Envoie une commande de découverte SDI-12 sur le bus, et lit les informations de tous les capteurs connecté.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→discoverSingleSensor()

Envoie une commandde de découverte SDI-12 sur le bus, et lit les information du capteur reçue.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→getSensorInformation(sensorAddr)

Envoie une commande de d'information SDI-12 sur le bus, et renvois les informations du capteur appelé.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→get_advertisedValue()

Retourne la valeur courante du port SDI12 (pas plus de 6 caractères).

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→get_currentJob()

Retourne le nom du fichier de tâches actif en ce moment.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→get_errCount()

Retourne le nombre d'erreurs de communication détectées depuis la dernière mise à zéro.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→get_errorMessage()

Retourne le message correspondant à la dernière erreur survenue lors de l'utilisation du port SDI12.

cpp m pas vb cs java py php ts es

sdi12port→get_errorType()

Retourne le code d'erreur correspondant à la dernière erreur survenue lors de l'utilisation du port SDI12.

cpp m pas vb cs java py php ts es

sdi12port→get_friendlyName()

Retourne un identifiant global du port SDI12 au format `NOM_MODULE.NOM_FONCTION`.

cpp m cs java py php ts es dnp

sdi12port→get_functionDescriptor()

Retourne un identifiant unique de type `YFUN_DESCR` correspondant à la fonction.

cpp m pas vb cs java py php ts es

sdi12port→get_functionId()

Retourne l'identifiant matériel du port SDI12, sans référence au module.

cpp m vb cs java py php ts es dnp

sdi12port→get_hardwareId()

Retourne l'identifiant matériel unique du port SDI12 au format `SERIAL.FUNCTIONID`.

cpp m vb cs java py php ts es dnp

sdi12port→get_jobMaxSize()

Retourne la taille maximale d'un fichier job.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→get_jobMaxTask()

Retourne le nombre maximal de tâches dans un job supporté par le module.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→get_lastMsg()

Retourne le dernier message reçu.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→get_logicalName()

Retourne le nom logique du port SDI12.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→get_module()

Retourne l'objet `YModule` correspondant au module Yoctopuce qui héberge la fonction.

cpp m pas vb cs java py php ts es dnp

sdi12port→get_module_async(callback, context)

Retourne l'objet `YModule` correspondant au module Yoctopuce qui héberge la fonction.

sdi12port→get_protocol()

Retourne le type de protocole utilisé sur la communication série, sous forme d'une chaîne de caractères.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→get_rxCount()

Retourne le nombre d'octets reçus depuis la dernière mise à zéro.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→get_rxMsgCount()

Retourne le nombre de messages reçus depuis la dernière mise à zéro.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→get_serialMode()

Retourne les paramètres de communication du port, sous forme d'une chaîne de caractères du type "1200,7E1,Simplex".

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→get_serialNumber()

Retourne le numéro de série du module, préprogrammé en usine.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→get_startupJob()

Retourne le nom du fichier de tâches à exécuter au démarrage du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→get_txCount()

Retourne le nombre d'octets transmis depuis la dernière mise à zéro.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→get_txMsgCount()

Retourne le nombre de messages envoyés depuis la dernière mise à zéro.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→get_userData()

Retourne le contenu de l'attribut userData, précédemment stocké à l'aide de la méthode `set_userData`.

cpp m pas vb cs java py php ts es

sdi12port→get_voltageLevel()

Retourne le niveau de tension utilisé par le module sur le port série.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→isOnline()

Vérifie si le module hébergeant le port SDI12 est joignable, sans déclencher d'erreur.

cpp m pas vb cs java py php ts es dnp

sdi12port→isOnline_async(callback, context)

Vérifie si le module hébergeant le port SDI12 est joignable, sans déclencher d'erreur.

sdi12port→isReadOnly()

Indique si la fonction est en lecture seule.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→load(msValidity)

Met en cache les valeurs courantes du port SDI12, avec une durée de validité spécifiée.

cpp m pas vb cs java py php ts es

sdi12port→loadAttribute(attrName)

Retourne la valeur actuelle d'un attribut spécifique de la fonction, sous forme de texte, le plus rapidement possible mais sans passer par le cache.

cpp m pas vb cs java uwp py php ts es dnp

sdi12port→load_async(msValidity, callback, context)

Met en cache les valeurs courantes du port SDI12, avec une durée de validité spécifiée.

sdi12port→muteValueCallbacks()

Désactive l'envoi de chaque changement de la valeur publiée au hub parent.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→nextSdi12Port()

Continue l'énumération des ports SDI12 commencée à l'aide de `yFirstSdi12Port()` Attention, vous ne pouvez faire aucune supposition sur l'ordre dans lequel les ports SDI12 sont retournés.

cpp m pas vb cs java uwp py php ts es

sdi12port→queryHex(hexString, maxWait)

Envoie un message binaire sur le port série, et lit la réponse reçue.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→queryLine(query, maxWait)

Envoie un message sous forme de ligne de texte sur le port série, et lit la réponse reçue.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→querySdi12(sensorAddr, cmd, maxWait)

Envoie une requête SDI-12 sur le bus, et lit la réponse immédiate reçue.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→readArray(nChars)

Lit le contenu du tampon de réception sous forme de liste d'octets, à partir de la position courante dans le flux de donnée.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→readBin(nChars)

Lit le contenu du tampon de réception sous forme d'objet binaire, à partir de la position courante dans le flux de donnée.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→readByte()

Lit le prochain byte dans le tampon de réception, à partir de la position courante dans le flux de donnée.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→readConcurrentMeasurements(sensorAddr)

Envoie une commande de lecture sur le bus, et renvois les informations du capteur appelé.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→readHex(nBytes)

Lit le contenu du tampon de réception sous forme hexadécimale, à partir de la position courante dans le flux de donnée.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→readLine()

Lit la prochaine ligne (ou le prochain message) du tampon de réception, à partir de la position courante dans le flux de donnée.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→readMessages(pattern, maxWait)

Cherche les messages entrants dans le tampon de réception correspondant à un format donné, à partir de la position courante.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→readSensor(sensorAddr, measCmd, maxWait)

Envoie une commande de mesure SDI-12 sur le bus, et lit la réponse immédiate reçue.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→readStr(nChars)

Lit le contenu du tampon de réception sous forme de string, à partir de la position courante dans le flux de donnée.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→read_avail()

Retourne le nombre de bytes prêts à être lus dans le tampon de réception, depuis la position courante dans le flux de donnée utilisé par l'objet d'API.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→read_seek(absPos)

Change le pointeur de position courante dans le flux de donnée à la valeur spécifiée.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→read_tell()

Retourne la valeur actuelle du pointeur de position courante dans le flux de donnée utilisé par l'objet d'API.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→registerValueCallback(callback)

Enregistre la fonction de callback qui est appelée à chaque changement de la valeur publiée.

cpp m pas vb cs java uwp py php ts es

sdi12port→requestConcurrentMeasurements(sensorAddr)

Envoie une commande de lecture sur le bus, et renvois les informations du capteur appelé.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→reset()

Remet à zéro tous les compteurs et efface les tampons.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→selectJob(jobfile)

Charge et execute le fichier de tâche spécifié.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→set_currentJob(newval)

Sélectionne un fichier de tâches pour exécution immédiate.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→set_logicalName(newval)

Modifie le nom logique du port SDI12.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→set_protocol(newval)

Modifie le type de protocol utilisé sur la communication série.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→set_serialMode(newval)

Modifie les paramètres de communication du port, sous forme d'une chaîne de caractères du type "1200,7E1,Simplex".

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→set_startupJob(newval)

Modifie le nom du job à exécuter au démarrage du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→set_userData(data)

Enregistre un contexte libre dans l'attribut userData de la fonction, afin de le retrouver plus tard à l'aide de la méthode `get_userData`.

cpp m pas vb cs java py php ts es

sdi12port→set_voltageLevel(newval)

Modifie le niveau de tension utilisé par le module sur le port série.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→snoopMessages(maxWait)

Récupère les messages dans la mémoire tampon du module (dans les deux directions), à partir de la position courante.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→snoopMessagesEx(maxWait, maxMsg)

Récupère les messages dans la mémoire tampon du module (dans les deux directions), à partir de la position courante.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→unmuteValueCallbacks()

Réactive l'envoi de chaque changement de la valeur publiée au hub parent.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→uploadJob(jobfile, jsonDef)

Sauvegarde une définition de tâche (au format JSON) dans un fichier.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→wait_async(callback, context)

Attend que toutes les commandes asynchrones en cours d'exécution sur le module soient terminées, et appelle le callback passé en paramètre.

ts es

sdi12port→writeArray(byteList)

Envoie une séquence d'octets (fournie sous forme d'une liste) sur le port série.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→writeBin(buff)

Envoie un objet binaire tel quel sur le port série.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→writeByte(code)

Envoie un unique byte sur le port série.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→writeHex(hexString)

Envoie une séquence d'octets (fournie sous forme de chaîne hexadécimale) sur le port série.

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→writeLine(text)

Envoie une chaîne de caractères sur le port série, suivie d'un saut de ligne (CR LF).

cpp m pas vb cs java uwp py php ts es dnp cmd

sdi12port→writeStr(text)

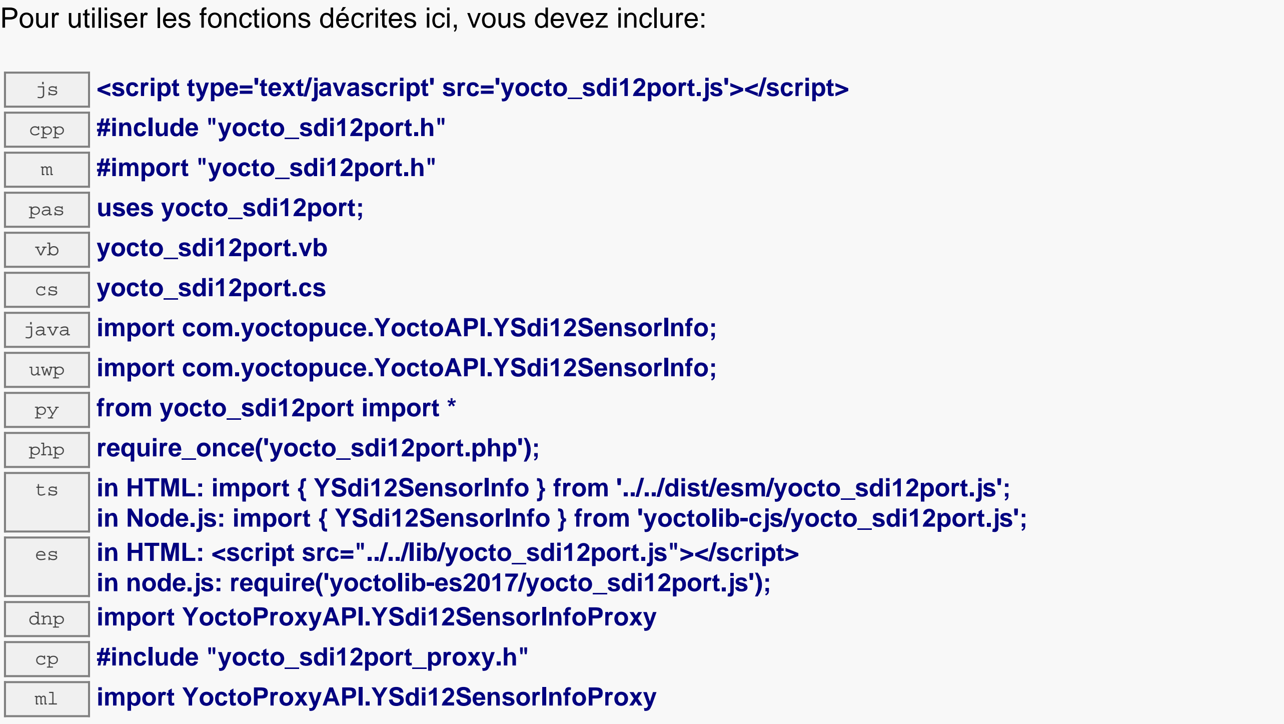
Envoie une chaîne de caractères telle quelle sur le port série.

cpp m pas vb cs java uwp py php ts es dnp cmd

25.4. La classe YSdi12SensorInfo

Description d'un capteur SDI12 détecté, retournée par les méthodes `sdi12Port.discoverSingleSensor` et `sdi12Port.discoverAllSensors`

Pour utiliser les fonctions décrites ici, vous devez inclure:



js	<script type='text/javascript' src='yocto_sdi12port.js'></script>
cpp	#include "yocto_sdi12port.h"
m	#import "yocto_sdi12port.h"
pas	uses yocto_sdi12port;
vb	yocto_sdi12port.vb
cs	yocto_sdi12port.cs
java	import com.yoctopuce.YoctoAPI.YSdi12SensorInfo;
uwp	import com.yoctopuce.YoctoAPI.YSdi12SensorInfo;
py	from yocto_sdi12port import *
php	require_once('yocto_sdi12port.php');
ts	in HTML: import { YSdi12SensorInfo } from '../../dist/esm/yocto_sdi12port.js'; in Node.js: import { YSdi12SensorInfo } from 'yoctolib-cjs/yocto_sdi12port.js';
es	in HTML: <script src="../../lib/yocto_sdi12port.js"></script> in node.js: require('yoctolib-es2017/yocto_sdi12port.js');
dnp	import YoctoProxyAPI.YSdi12SensorInfoProxy
cp	#include "yocto_sdi12port_proxy.h"
ml	import YoctoProxyAPI.YSdi12SensorInfoProxy

Propriétés des objets `YSdi12SensorInfoProxy`

sdi12sensorinfo→IsValid ***[lecture seule]***

Etat du capteur.

dnp

sdi12sensorinfo→MeasureCount ***[lecture seule]***

Nombre de mesure du capteur.

dnp

sdi12sensorinfo→SensorAddress ***[lecture seule]***

Adresse du capteur.

dnp

sdi12sensorinfo→SensorModel ***[lecture seule]***

Numéro de modèle du capteur.

dnp

sdi12sensorinfo→SensorProtocol ***[lecture seule]***

Version SDI-12 compatible du capteur.

dnp

sdi12sensorinfo→SensorSerial ***[lecture seule]***

Numéro de série du capteur.

dnp

sdi12sensorinfo→SensorVendor ***[lecture seule]***

Identification du vendeur du capteur.

dnp

sdi12sensorinfo→SensorVersion *[lecture seule]*

Version du capteur.

dnp

Méthodes des objets `YSdi12SensorInfo`

sdi12sensorinfo→get_measureCommand(measureIndex)

Retourne la commande de mesure du capteur.

cpp m pas vb cs java uwp py php ts es dnp

sdi12sensorinfo→get_measureCount()

Retourne le nombre de mesure du capteur.

cpp m pas vb cs java uwp py php ts es dnp

sdi12sensorinfo→get_measureDescription(measureIndex)

Retourne la description de la valeur mesurée.

cpp m pas vb cs java uwp py php ts es dnp

sdi12sensorinfo→get_measurePosition(measureIndex)

Retourne la position de mesure du capteur.

cpp m pas vb cs java uwp py php ts es dnp

sdi12sensorinfo→get_measureSymbol(measureIndex)

Retourne le symbole de la valeur mesurée.

cpp m pas vb cs java uwp py php ts es dnp

sdi12sensorinfo→get_measureUnit(measureIndex)

Retourne l'unité de la valeur mesurée.

cpp m pas vb cs java uwp py php ts es dnp

sdi12sensorinfo→get_sensorAddress()

Retourne l'adresse du capteur.

cpp m pas vb cs java uwp py php ts es dnp

sdi12sensorinfo→get_sensorModel()

Retourne le numéro de modèle du capteur.

cpp m pas vb cs java uwp py php ts es dnp

sdi12sensorinfo→get_sensorProtocol()

Retourne la version SDI-12 compatible du capteur.

cpp m pas vb cs java uwp py php ts es dnp

sdi12sensorinfo→get_sensorSerial()

Retourne le numéro de série du capteur.

cpp m pas vb cs java uwp py php ts es dnp

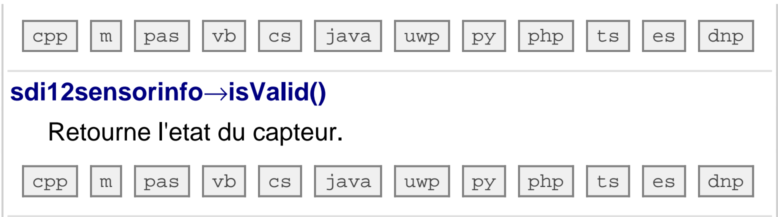
sdi12sensorinfo→get_sensorVendor()

Retourne l'identification du vendeur du capteur.

cpp m pas vb cs java uwp py php ts es dnp

sdi12sensorinfo→get_sensorVersion()

Retourne la version du capteur.



cpp m pas vb cs java uwp py php ts es dnp

sdi12sensorinfo→isValid()

Retourne l'etat du capteur.

cpp m pas vb cs java uwp py php ts es dnp

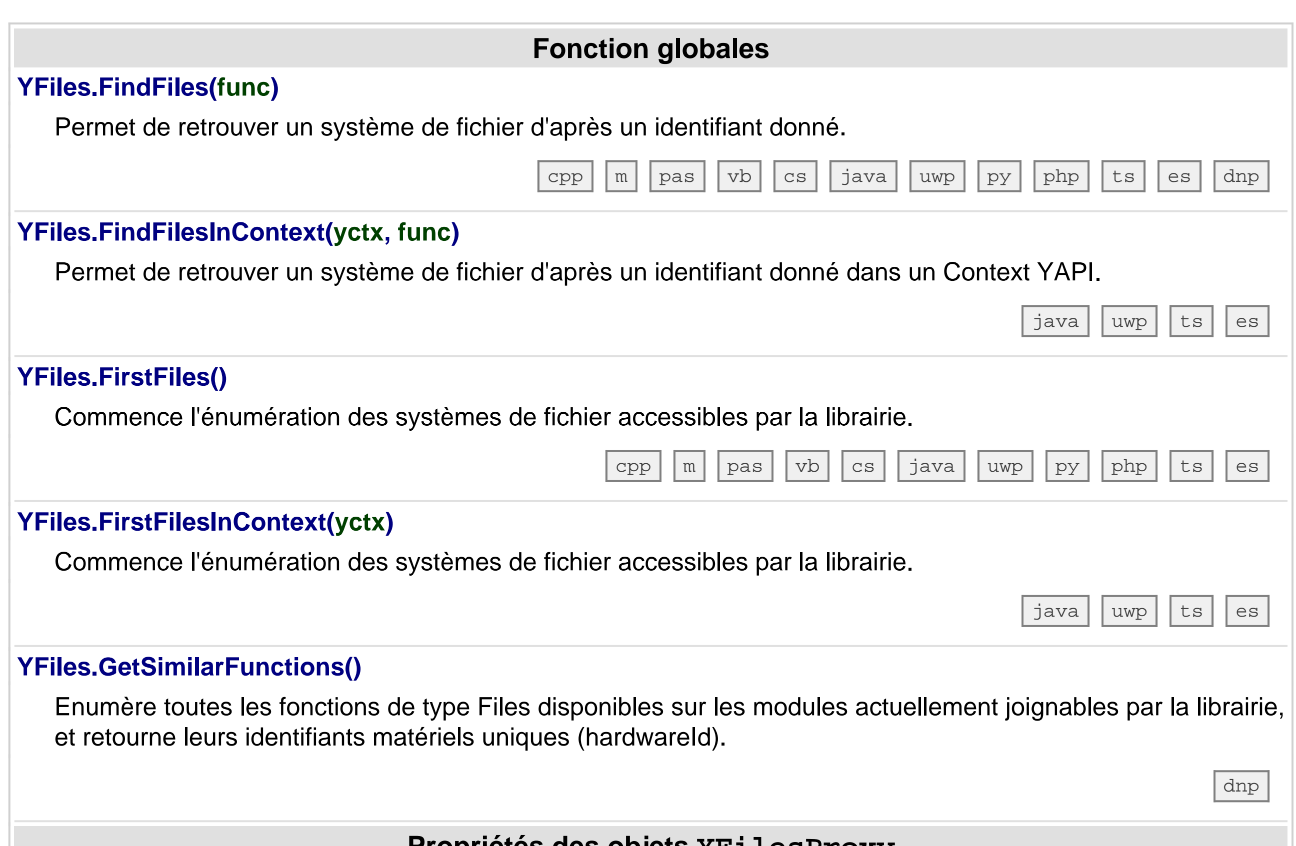
25.5. La classe YFiles

Interface pour intéragir avec les systèmes de fichier, disponibles par exemple dans le Yocto-Color-V2, le Yocto-SPI, le YoctoHub-Ethernet et le YoctoHub-GSM-4G

La class YFiles permet d'accéder au système de fichier embarqué sur certains modules Yoctopuce. Le stockage de fichiers permet par exemple de personnaliser un service web (dans le cas d'un module connecté au réseau) ou pour d'ajouter un police de caractères (dans le cas d'un module d'affichage).

Pour utiliser les fonctions décrites ici, vous devez inclure:

js	**<script type='text/javascript' src='yocto_files.js'></script>**
cpp	**#include "yocto_files.h"**
m	**#import "yocto_files.h"**
pas	**uses yocto_files;**
vb	**yocto_files.vb**
cs	**yocto_files.cs**
java	**import com.yoctopuce.YoctoAPI.YFiles;**
uwp	**import com.yoctopuce.YoctoAPI.YFiles;**
py	**from yocto_files import ***
php	**require_once('yocto_files.php');**
ts	**in HTML: import { YFiles } from '../../dist/esm/yocto_files.js';** **in Node.js: import { YFiles } from 'yoctolib-cjs/yocto_files.js';**
es	**in HTML: <script src="../../lib/yocto_files.js"></script>** **in node.js: require('yoctolib-es2017/yocto_files.js');**
dnp	**import YoctoProxyAPI.YFilesProxy**
cp	**#include "yocto_files_proxy.h"**
vi	**YFiles.vi**
ml	**import YoctoProxyAPI.YFilesProxy**



Fonction globales

YFiles.FindFiles(func)

Permet de retrouver un système de fichier d'après un identifiant donné.

cpp m pas vb cs java uwp py php ts es dnp

YFiles.FindFilesInContext(yctx, func)

Permet de retrouver un système de fichier d'après un identifiant donné dans un Context YAPI.

java uwp ts es

YFiles.FirstFiles()

Commence l'énumération des systèmes de fichier accessibles par la librairie.

cpp m pas vb cs java uwp py php ts es

YFiles.FirstFilesInContext(yctx)

Commence l'énumération des systèmes de fichier accessibles par la librairie.

java uwp ts es

YFiles.GetSimilarFunctions()

Enumère toutes les fonctions de type Files disponibles sur les modules actuellement joignables par la librairie, et retourne leurs identifiants matériels uniques (hardwareId).

dnp

Propriétés des objets `YFilesProxy`

files→AdvertisedValue *[lecture seule]*

Courte chaîne de caractères représentant l'état courant de la fonction.

dnp

files→FilesCount *[lecture seule]*

Nombre de fichiers présents dans le système de fichier.

dnp

files→FriendlyName *[lecture seule]*

Identifiant global de la fonction au format `NOM_MODULE.NOM_FONCTION`.

dnp

files→FunctionId *[lecture seule]*

Identifiant matériel du système de fichier, sans référence au module.

dnp

files→HardwareId *[lecture seule]*

Identifiant matériel unique de la fonction au format `SERIAL.FUNCTIONID`.

dnp

files→IsOnline *[lecture seule]*

Vérifie si le module hébergeant la fonction est joignable, sans déclencher d'erreur.

dnp

files→LogicalName *[modifiable]*

Nom logique de la fonction.

dnp

files→SerialNumber *[lecture seule]*

Numéro de série du module, préprogrammé en usine.

dnp

Méthodes des objets `YFiles`

files→clearCache()

Invalide le cache.

cpp m pas vb cs java py php ts es

files→describe()

Retourne un court texte décrivant de manière non-ambigüe l'instance du système de fichier au format `TYPE(NAME)=SERIAL.FUNCTIONID`.

cpp m pas vb cs java py php ts es

files→download(pathname)

Télécharge le fichier choisi du filesystème et retourne son contenu.

cpp m pas vb cs java uwp py php ts es dnp cmd

files→download_async(pathname, callback, context)

Procède au chargement du bloc suivant de mesures depuis l'enregistreur de données du module, de manière asynchrone.

files→fileExist(filename)

Test si un fichier esit dans le système de fichier du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

files→format_fs()

Rétabli le système de fichier dans on état original, défragmenté.

cpp m pas vb cs java uwp py php ts es dnp cmd

files→get_advertisedValue()

Retourne la valeur courante du système de fichier (pas plus de 6 caractères).

cpp m pas vb cs java uwp py php ts es dnp cmd

files→get_errorMessage()

Retourne le message correspondant à la dernière erreur survenue lors de l'utilisation du système de fichier.

cpp m pas vb cs java py php ts es

files→get_errorType()

Retourne le code d'erreur correspondant à la dernière erreur survenue lors de l'utilisation du système de fichier.

cpp m pas vb cs java py php ts es

files→get_filesCount()

Retourne le nombre de fichiers présents dans le système de fichier.

cpp m pas vb cs java uwp py php ts es dnp cmd

files→get_freeSpace()

Retourne l'espace disponible dans le système de fichier pour charger des nouveaux fichiers, en octets.

cpp m pas vb cs java uwp py php ts es dnp cmd

files→get_friendlyName()

Retourne un identifiant global du système de fichier au format `NOM_MODULE.NOM_FONCTION`.

cpp m cs java py php ts es dnp

files→get_functionDescriptor()

Retourne un identifiant unique de type `YFUN_DESCR` correspondant à la fonction.

cpp m pas vb cs java py php ts es

files→get_functionId()

Retourne l'identifiant matériel du système de fichier, sans référence au module.

cpp m vb cs java py php ts es dnp

files→get_hardwareId()

Retourne l'identifiant matériel unique du système de fichier au format `SERIAL.FUNCTIONID`.

cpp m vb cs java py php ts es dnp

files→get_list(pattern)

Retourne une liste d'objets objet YFileRecord qui décrivent les fichiers présents dans le système de fichier.

cpp m pas vb cs java uwp py php ts es dnp cmd

files→get_logicalName()

Retourne le nom logique du système de fichier.

cpp m pas vb cs java uwp py php ts es dnp cmd

files→get_module()

Retourne l'objet `YModule` correspondant au module Yoctopuce qui héberge la fonction.

cpp m pas vb cs java py php ts es dnp

files→get_module_async(callback, context)

Retourne l'objet `YModule` correspondant au module Yoctopuce qui héberge la fonction.

files→get_serialNumber()

Retourne le numéro de série du module, préprogrammé en usine.

cpp m pas vb cs java uwp py php ts es dnp cmd

files→get_userData()

Retourne le contenu de l'attribut userData, précédemment stocké à l'aide de la méthode `set_userData`.

cpp m pas vb cs java py php ts es

files→isOnline()

Vérifie si le module hébergeant le système de fichier est joignable, sans déclencher d'erreur.

cpp m pas vb cs java py php ts es dnp

files→isOnline_async(callback, context)

Vérifie si le module hébergeant le système de fichier est joignable, sans déclencher d'erreur.

files→isReadOnly()

Indique si la fonction est en lecture seule.

cpp m pas vb cs java uwp py php ts es dnp cmd

files→load(msValidity)

Met en cache les valeurs courantes du système de fichier, avec une durée de validité spécifiée.

cpp m pas vb cs java py php ts es

files→loadAttribute(attrName)

Retourne la valeur actuelle d'un attribut spécifique de la fonction, sous forme de texte, le plus rapidement possible mais sans passer par le cache.

cpp m pas vb cs java uwp py php ts es dnp

files→load_async(msValidity, callback, context)

Met en cache les valeurs courantes du système de fichier, avec une durée de validité spécifiée.

files→muteValueCallbacks()

Désactive l'envoi de chaque changement de la valeur publiée au hub parent.

cpp m pas vb cs java uwp py php ts es dnp cmd

files→nextFiles()

Continue l'énumération des systèmes de fichier commencée à l'aide de `yFirstFiles()` Attention, vous ne pouvez faire aucune supposition sur l'ordre dans lequel les systèmes de fichier sont retournés.

cpp m pas vb cs java uwp py php ts es

files→registerValueCallback(callback)

Enregistre la fonction de callback qui est appelée à chaque changement de la valeur publiée.

cpp m pas vb cs java uwp py php ts es

files→remove(pathname)

Efface un fichier, spécifié par son path complet, du système de fichier.

cpp m pas vb cs java uwp py php ts es dnp cmd

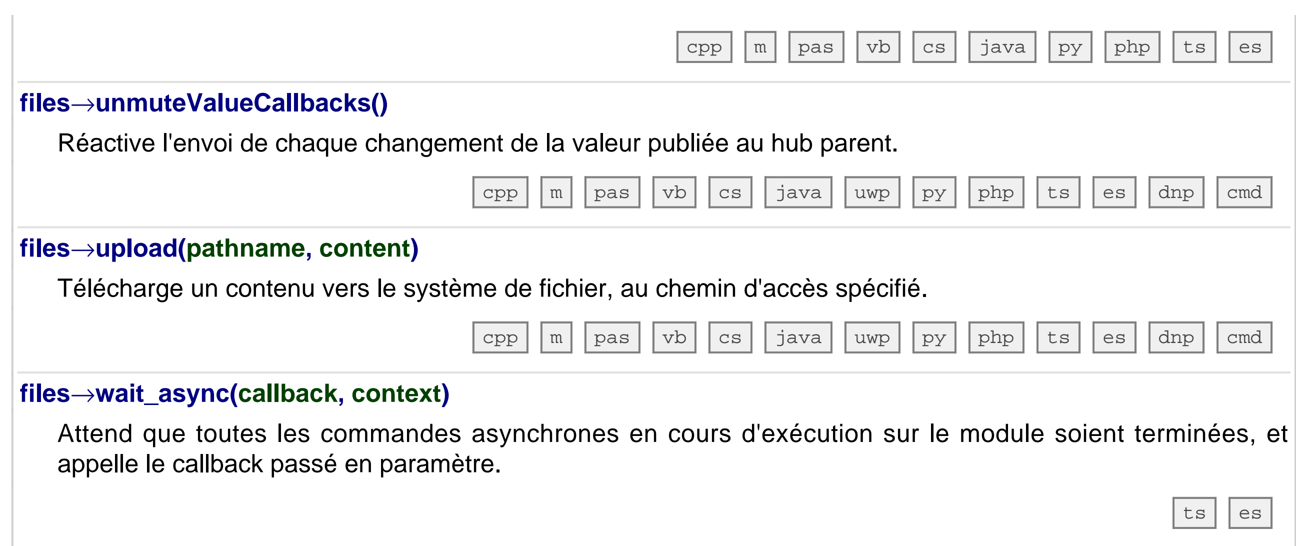
files→set_logicalName(newval)

Modifie le nom logique du système de fichier.

cpp m pas vb cs java uwp py php ts es dnp cmd

files→set_userData(data)

Enregistre un contexte libre dans l'attribut userData de la fonction, afin de le retrouver plus tard à l'aide de la méthode `get_userData`.



files→unmuteValueCallbacks()

Réactive l'envoi de chaque changement de la valeur publiée au hub parent.

cpp m pas vb cs java uwp py php ts es dnp cmd

files→upload(pathname, content)

Télécharge un contenu vers le système de fichier, au chemin d'accès spécifié.

cpp m pas vb cs java uwp py php ts es dnp cmd

files→wait_async(callback, context)

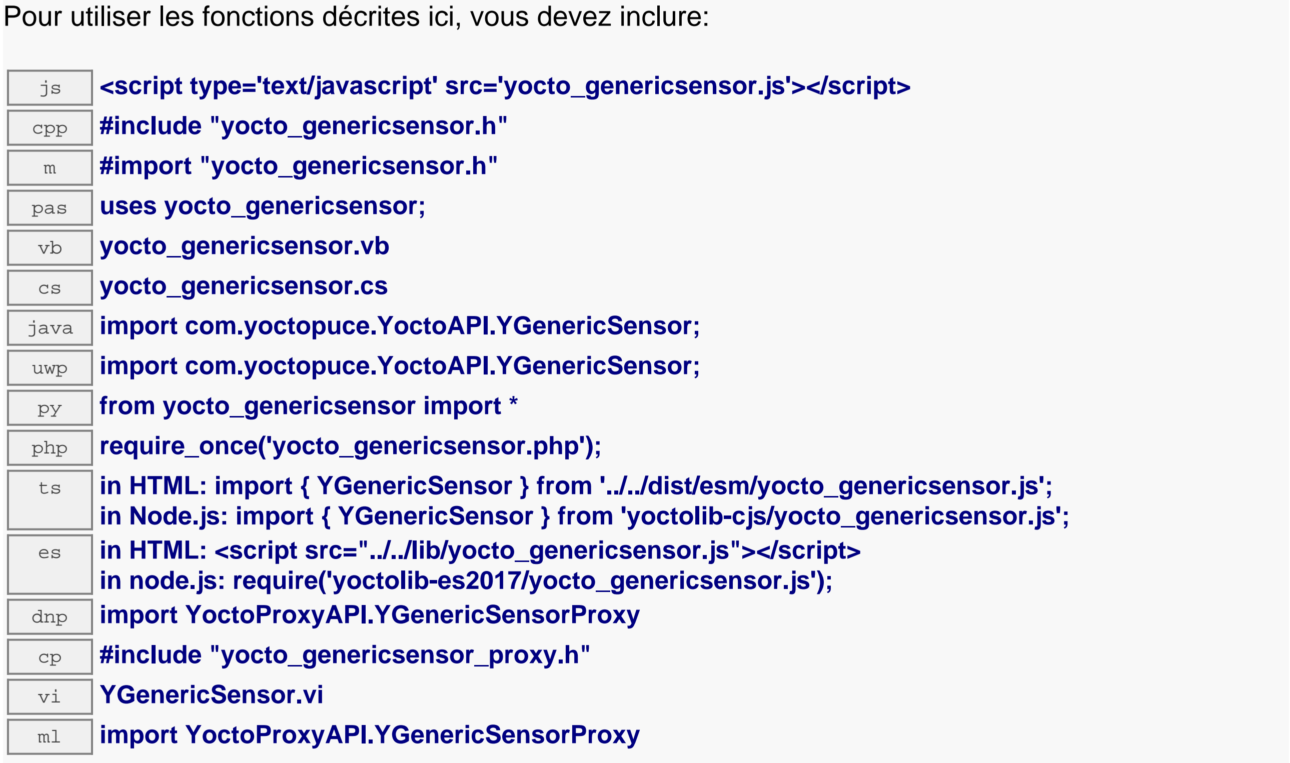
Attend que toutes les commandes asynchrones en cours d'exécution sur le module soient terminées, et appelle le callback passé en paramètre.

ts es

25.6. La classe YGenericSensor

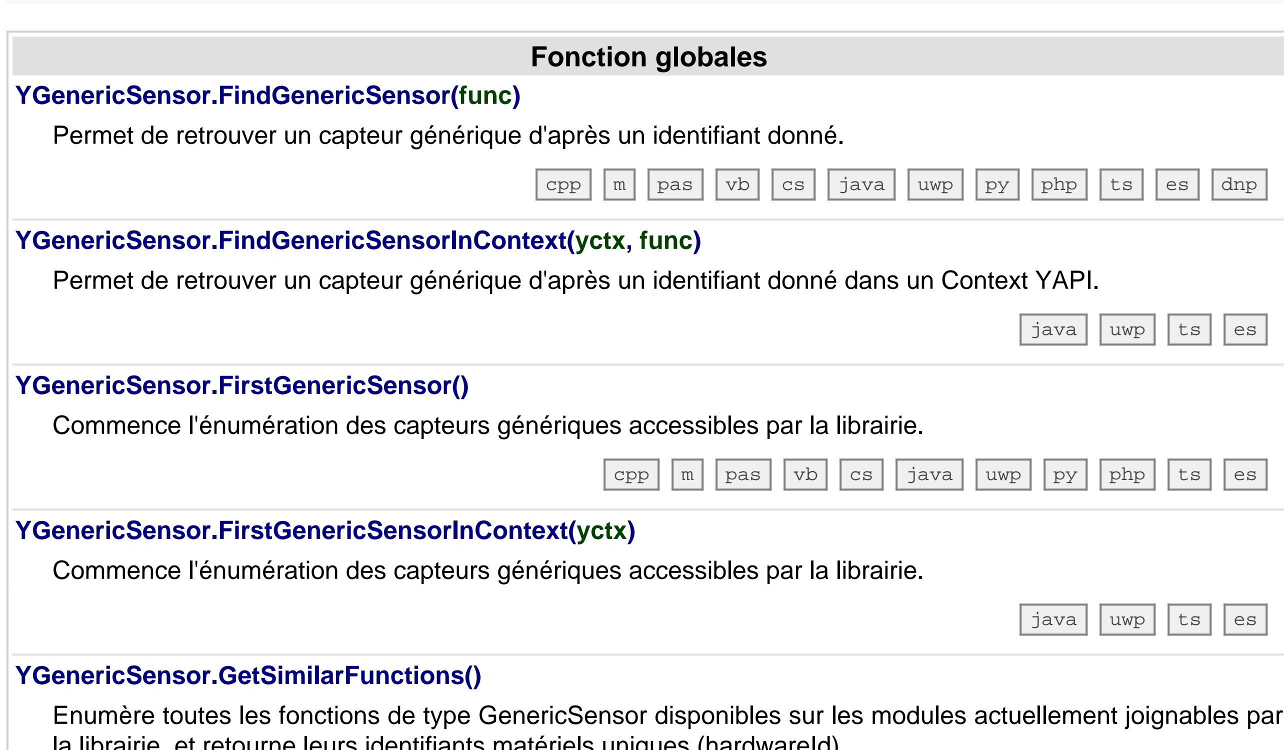
Interface pour intéragir avec les capteurs de type GenericSensor, disponibles par exemple dans le Yocto-0-10V-Rx, le Yocto-4-20mA-Rx, le Yocto-Bridge et le Yocto-milliVolt-Rx

La classe `YGenericSensor` permet de lire et de configurer les transducteurs de signaux Yoctopuce. Elle hérite de la classe `YSensor` toutes les fonctions de base des capteurs Yoctopuce: lecture de mesures, callbacks, enregistreur de données. De plus, elle permet de configurer une conversion automatique entre le signal mesuré et la grandeur physique représentée.



Pour utiliser les fonctions décrites ici, vous devez inclure:

js	<script type='text/javascript' src='yocto_genericsensor.js'></script>
cpp	#include "yocto_genericsensor.h"
m	#import "yocto_genericsensor.h"
pas	uses yocto_genericsensor;
vb	yocto_genericsensor.vb
cs	yocto_genericsensor.cs
java	import com.yoctopuce.YoctoAPI.YGenericSensor;
uwp	import com.yoctopuce.YoctoAPI.YGenericSensor;
py	from yocto_genericsensor import *
php	require_once('yocto_genericsensor.php');
ts	in HTML: import { YGenericSensor } from '../../dist/esm/yocto_genericsensor.js'; in Node.js: import { YGenericSensor } from 'yoctolib-cjs/yocto_genericsensor.js';
es	in HTML: <script src="../../lib/yocto_genericsensor.js"></script> in node.js: require('yoctolib-es2017/yocto_genericsensor.js');
dnp	import YoctoProxyAPI.YGenericSensorProxy
cp	#include "yocto_genericsensor_proxy.h"
vi	YGenericSensor.vi
ml	import YoctoProxyAPI.YGenericSensorProxy



Fonction globales

YGenericSensor.FindGenericSensor(func)

Permet de retrouver un capteur générique d'après un identifiant donné.

cpp m pas vb cs java uwp py php ts es dnp

YGenericSensor.FindGenericSensorInContext(yctx, func)

Permet de retrouver un capteur générique d'après un identifiant donné dans un Context YAPI.

java uwp ts es

YGenericSensor.FirstGenericSensor()

Commence l'énumération des capteurs génériques accessibles par la librairie.

cpp m pas vb cs java uwp py php ts es

YGenericSensor.FirstGenericSensorInContext(yctx)

Commence l'énumération des capteurs génériques accessibles par la librairie.

java uwp ts es

YGenericSensor.GetSimilarFunctions()

Enumère toutes les fonctions de type GenericSensor disponibles sur les modules actuellement joignables par la librairie, et retourne leurs identifiants matériels uniques (hardwareId).

dnp

Propriétés des objets `YGenericSensorProxy`

genericsensor→AdvMode *[modifiable]*

Mode de calcul de la valeur publiée jusqu'au hub parent (advertisedValue).

`dnp`

genericsensor→AdvertisedValue *[lecture seule]*

Courte chaîne de caractères représentant l'état courant de la fonction.

`dnp`

genericsensor→Enabled *[modifiable]*

état d'activation de cette mesure.

`dnp`

genericsensor→FriendlyName *[lecture seule]*

Identifiant global de la fonction au format `NOM_MODULE.NOM_FONCTION`.

`dnp`

genericsensor→FunctionId *[lecture seule]*

Identifiant matériel du senseur, sans référence au module.

`dnp`

genericsensor→HardwareId *[lecture seule]*

Identifiant matériel unique de la fonction au format `SERIAL.FUNCTIONID`.

`dnp`

genericsensor→IsOnline *[lecture seule]*

Vérifie si le module hébergeant la fonction est joignable, sans déclencher d'erreur.

`dnp`

genericsensor→LogFrequency *[modifiable]*

Fréquence d'enregistrement des mesures dans le datalogger, ou "OFF" si les mesures ne sont pas stockées dans la mémoire de l'enregistreur de données.

`dnp`

genericsensor→LogicalName *[modifiable]*

Nom logique de la fonction.

`dnp`

genericsensor→ReportFrequency *[modifiable]*

Fréquence de notification périodique des valeurs mesurées, ou "OFF" si les notifications périodiques sont désactivées pour cette fonction.

`dnp`

genericsensor→Resolution *[modifiable]*

Résolution des valeurs mesurées.

`dnp`

genericsensor→SerialNumber *[lecture seule]*

Numéro de série du module, préprogrammé en usine.

`dnp`

genericsensor→SignalBias *[modifiable]*

Biais du signal électrique pour la correction du point zéro.

dnp

genericsensor→SignalRange *[modifiable]*

Plage de signal d'entrée utilisé par le capteur.

dnp

genericsensor→SignalSampling *[modifiable]*

Méthode d'échantillonnage du signal utilisée.

dnp

genericsensor→SignalUnit *[lecture seule]*

Unité du signal électrique utilisé par le capteur.

dnp

genericsensor→ValueRange *[modifiable]*

Plage de valeurs physiques mesurés par le capteur.

dnp

Méthodes des objets `YGenericSensor`

genericsensor→calibrateFromPoints(rawValues, refValues)

Enregistre des points de correction de mesure, typiquement pour compenser l'effet d'un boîtier sur les mesures rendues par le capteur.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→clearCache()

Invalide le cache.

cpp m pas vb cs java py php ts es

genericsensor→describe()

Retourne un court texte décrivant de manière non-ambigüe l'instance du capteur générique au format `TYPE(NAME)=SERIAL.FUNCTIONID`.

cpp m pas vb cs java py php ts es

genericsensor→get_advMode()

Retourne le mode de calcul de la valeur publiée jusqu'au hub parent (advertisedValue).

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→get_advertisedValue()

Retourne la valeur courante du capteur générique (pas plus de 6 caractères).

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→get_currentRawValue()

Retourne la valeur brute retournée par le capteur (sans arrondi ni calibration).

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→get_currentValue()

Retourne la valeur mesurée actuelle.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→get_dataLogger()

Retourne l'objet `YDatalogger` du module qui héberge le senseur.

cpp m pas vb cs java uwp py php ts es dnp

genericsensor→get_enabled()

Retourne l'état d'activation de cette mesure.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→get_errorMessage()

Retourne le message correspondant à la dernière erreur survenue lors de l'utilisation du capteur générique.

cpp m pas vb cs java py php ts es

genericsensor→get_errorType()

Retourne le code d'erreur correspondant à la dernière erreur survenue lors de l'utilisation du capteur générique.

cpp m pas vb cs java py php ts es

genericsensor→get_friendlyName()

Retourne un identifiant global du capteur générique au format `NOM_MODULE.NOM_FONCTION`.

cpp m cs java py php ts es dnp

genericsensor→get_functionDescriptor()

Retourne un identifiant unique de type `YFUN_DESCR` correspondant à la fonction.

cpp m pas vb cs java py php ts es

genericsensor→get_functionId()

Retourne l'identifiant matériel du capteur générique, sans référence au module.

cpp m vb cs java py php ts es dnp

genericsensor→get_hardwareId()

Retourne l'identifiant matériel unique du capteur générique au format `SERIAL.FUNCTIONID`.

cpp m vb cs java py php ts es dnp

genericsensor→get_highestValue()

Retourne la valeur maximale observée pour la mesure depuis le démarrage du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→get_logFrequency()

Retourne la fréquence d'enregistrement des mesures dans le datalogger, ou "OFF" si les mesures ne sont pas stockées dans la mémoire de l'enregistreur de données.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→get_logicalName()

Retourne le nom logique du capteur générique.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→get_lowestValue()

Retourne la valeur minimale observée pour la mesure depuis le démarrage du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→get_module()

Retourne l'objet `YModule` correspondant au module Yoctopuce qui héberge la fonction.

cpp m pas vb cs java py php ts es dnp

genericsensor→get_module_async(callback, context)

Retourne l'objet `YModule` correspondant au module Yoctopuce qui héberge la fonction.

genericsensor→get_recordedData(startTime, endTime)

Retourne un objet `YDataSet` représentant des mesures de ce capteur précédemment enregistrées à l'aide du DataLogger, pour l'intervalle de temps spécifié.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→get_reportFrequency()

Retourne la fréquence de notification périodique des valeurs mesurées, ou "OFF" si les notifications périodiques sont désactivées pour cette fonction.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→get_resolution()

Retourne la résolution des valeurs mesurées.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→get_sensorState()

Retourne le code d'état du capteur, qui vaut zéro lorsqu'une mesure actuelle est disponible, ou un code positif si le capteur n'est pas en mesure de fournir une valeur en ce moment.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→get_serialNumber()

Retourne le numéro de série du module, préprogrammé en usine.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→get_signalBias()

Retourne le biais du signal électrique pour la correction du point zéro.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→get_signalRange()

Retourne la plage de signal d'entrée utilisé par le capteur.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→get_signalSampling()

Retourne la méthode d'échantillonnage du signal utilisée.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→get_signalUnit()

Retourne l'unité du signal électrique utilisé par le capteur.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→get_signalValue()

Retourne la valeur actuelle du signal électrique mesuré par le capteur.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→get_unit()

Retourne l'unité dans laquelle la mesure est exprimée.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→get_userData()

Retourne le contenu de l'attribut userData, précédemment stocké à l'aide de la méthode `set_userData`.

cpp m pas vb cs java py php ts es

genericsensor→get_valueRange()

Retourne la plage de valeurs physiques mesurés par le capteur.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→isOnline()

Vérifie si le module hébergeant le capteur générique est joignable, sans déclencher d'erreur.

cpp m pas vb cs java py php ts es dnp

genericsensor→isOnline_async(callback, context)

Vérifie si le module hébergeant le capteur générique est joignable, sans déclencher d'erreur.

genericsensor→isReadOnly()

Indique si la fonction est en lecture seule.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→isSensorReady()

Vérifie si le capteur est actuellement en état de transmettre une mesure valide.

cmd

genericsensor→load(msValidity)

Met en cache les valeurs courantes du capteur générique, avec une durée de validité spécifiée.

cpp m pas vb cs java py php ts es

genericsensor→loadAttribute(attrName)

Retourne la valeur actuelle d'un attribut spécifique de la fonction, sous forme de texte, le plus rapidement possible mais sans passer par le cache.

cpp m pas vb cs java uwp py php ts es dnp

genericsensor→loadCalibrationPoints(rawValues, refValues)

Récupère les points de correction de mesure précédemment enregistrés à l'aide de la méthode `calibrateFromPoints`.

cpp m pas vb cs java uwp py php ts es cmd

genericsensor→load_async(msValidity, callback, context)

Met en cache les valeurs courantes du capteur générique, avec une durée de validité spécifiée.

genericsensor→muteValueCallbacks()

Désactive l'envoi de chaque changement de la valeur publiée au hub parent.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→nextGenericSensor()

Continue l'énumération des capteurs génériques commencée à l'aide de `yFirstGenericSensor()` Attention, vous ne pouvez faire aucune supposition sur l'ordre dans lequel les capteurs génériques sont retournés.

cpp m pas vb cs java uwp py php ts es

genericsensor→registerTimedReportCallback(callback)

Enregistre la fonction de callback qui est appelée à chaque notification périodique.

cpp m pas vb cs java uwp py php ts es

genericsensor→registerValueCallback(callback)

Enregistre la fonction de callback qui est appelée à chaque changement de la valeur publiée.

cpp m pas vb cs java uwp py php ts es

genericsensor→set_advMode(newval)

Modifie le mode de calcul de la valeur publiée jusqu'au hub parent (advertisedValue).

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→set_enabled(newval)

Modifie l'état d'activation de cette mesure.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→set_highestValue(newval)

Modifie la mémoire de valeur maximale observée.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→set_logFrequency(newval)

Modifie la fréquence d'enregistrement des mesures dans le datalogger.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→set_logicalName(newval)

Modifie le nom logique du capteur générique.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→set_lowestValue(newval)

Modifie la mémoire de valeur minimale observée.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→set_reportFrequency(newval)

Modifie la fréquence de notification périodique des valeurs mesurées.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→set_resolution(newval)

Change la résolution des valeurs physique mesurées.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→set_signalBias(newval)

Modifie le biais du signal électrique pour la correction du point zéro.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→set_signalRange(newval)

Modifie la plage du signal d'entrée utilisé par le capteur.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→set_signalSampling(newval)

Modifie la méthode d'échantillonnage du signal à utiliser.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→set_unit(newval)

Modifie l'unité dans laquelle la valeur mesurée est exprimée.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→set_userData(data)

Enregistre un contexte libre dans l'attribut userData de la fonction, afin de le retrouver plus tard à l'aide de la méthode `get_userData`.

cpp m pas vb cs java py php ts es

genericsensor→set_valueRange(newval)

Modifie la plage de valeur de sortie, correspondant à la grandeur physique mesurée par le capteur.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→startDataLogger()

Démarre l'enregistreur de données du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→stopDataLogger()

Arrête l'enregistreur de données du module.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→unmuteValueCallbacks()

Réactive l'envoi de chaque changement de la valeur publiée au hub parent.

cpp m pas vb cs java uwp py php ts es dnp cmd

genericsensor→wait_async(callback, context)

Attend que toutes les commandes asynchrones en cours d'exécution sur le module soient terminées, et appelle le callback passé en paramètre.

ts es

genericsensor→zeroAdjust()

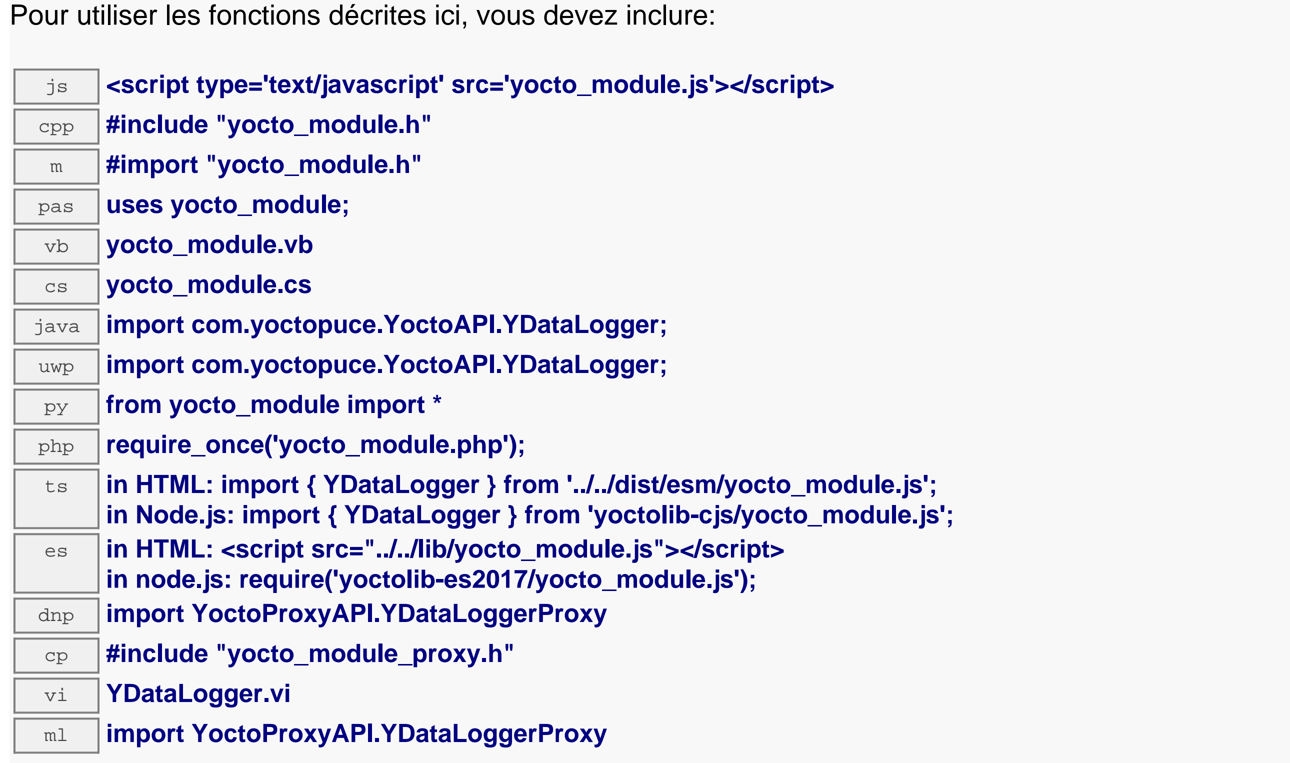
Ajuste le biais du signal de sorte à ce que la valeur actuelle du signal soit interprétée comme zéro (tare).

cpp m pas vb cs java uwp py php ts es dnp cmd

25.7. La classe YDataLogger

Interface de contrôle de l'enregistreur de données, présent sur la plupart des capteurs Yoctopuce.

La plupart des capteurs Yoctopuce sont équipés d'une mémoire non-volatile. Elle permet de mémoriser les données mesurées d'une manière autonome, sans nécessiter le suivi permanent d'un ordinateur. La classe `YDataLogger` contrôle les paramètres globaux de cet enregistreur de données. Le contrôle de l'enregistrement (start / stop) et la récupération des données se fait au niveau des objets qui gèrent les senseurs.



Pour utiliser les fonctions décrites ici, vous devez inclure:

js	<script type='text/javascript' src='yocto_module.js'></script>
cpp	#include "yocto_module.h"
m	#import "yocto_module.h"
pas	uses yocto_module;
vb	yocto_module.vb
cs	yocto_module.cs
java	import com.yoctopuce.YoctoAPI.YDataLogger;
uwp	import com.yoctopuce.YoctoAPI.YDataLogger;
py	from yocto_module import *
php	require_once('yocto_module.php');
ts	in HTML: import { YDataLogger } from '../../dist/esm/yocto_module.js'; in Node.js: import { YDataLogger } from 'yoctolib-cjs/yocto_module.js';
es	in HTML: <script src="../../lib/yocto_module.js"></script> in node.js: require('yoctolib-es2017/yocto_module.js');
dnp	import YoctoProxyAPI.YDataLoggerProxy
cp	#include "yocto_module_proxy.h"
vi	YDataLogger.vi
ml	import YoctoProxyAPI.YDataLoggerProxy

Fonction globales

YDataLogger.FindDataLogger(func)

Permet de retrouver un enregistreur de données d'après un identifiant donné.

cpp m pas vb cs java uwp py php ts es dnp

YDataLogger.FindDataLoggerInContext(yctx, func)

Permet de retrouver un enregistreur de données d'après un identifiant donné dans un Context YAPI.

java uwp ts es

YDataLogger.FirstDataLogger()

Commence l'énumération des enregistreurs de données accessibles par la librairie.

cpp m pas vb cs java uwp py php ts es

YDataLogger.FirstDataLoggerInContext(yctx)

Commence l'énumération des enregistreurs de données accessibles par la librairie.

java uwp ts es

YDataLogger.GetSimilarFunctions()

Enumère toutes les fonctions de type DataLogger disponibles sur les modules actuellement joignables par la librairie, et retourne leurs identifiants matériels uniques (hardwareId).

dnp

Propriétés des objets YDataLoggerProxy

datalogger→AdvertisedValue ***[lecture seule]***

Courte chaîne de caractères représentant l'état courant de la fonction.

dnp

datalogger→AutoStart ***[modifiable]***

Mode d'activation automatique de l'enregistreur de données à la mise sous tension.

dnp

datalogger→BeaconDriven ***[modifiable]***

Vrai si l'enregistreur de données est synchronisé avec la balise de localisation.

dnp

datalogger→FriendlyName ***[lecture seule]***

Identifiant global de la fonction au format `NOM_MODULE.NOM_FONCTION`.

dnp

datalogger→FunctionId ***[lecture seule]***

Identifiant matériel de l'enregistreur de données, sans référence au module.

dnp

datalogger→HardwareId ***[lecture seule]***

Identifiant matériel unique de la fonction au format `SERIAL.FUNCTIONID`.

dnp

datalogger→IsOnline ***[lecture seule]***

Vérifie si le module hébergeant la fonction est joignable, sans déclencher d'erreur.

dnp

datalogger→LogicalName ***[modifiable]***

Nom logique de la fonction.

dnp

datalogger→Recording ***[modifiable]***

état d'activation de l'enregistreur de données.

dnp

datalogger→SerialNumber ***[lecture seule]***

Numéro de série du module, préprogrammé en usine.

dnp

Méthodes des objets YDataLogger

datalogger→clearCache()

Invalide le cache.

cpp m pas vb cs java py php ts es

datalogger→describe()

Retourne un court texte décrivant de manière non-ambigüe l'instance de l'enregistreur de données au format `TYPE(NAME)=SERIAL.FUNCTIONID`.

cpp m pas vb cs java py php ts es

datalogger→forgetAllDataStreams()

Efface tout l'historique des mesures de l'enregistreur de données.

cpp m pas vb cs java uwp py php ts es dnp cmd

datalogger→get_advertisedValue()

Retourne la valeur courante de l'enregistreur de données (pas plus de 6 caractères).

cpp m pas vb cs java uwp py php ts es dnp cmd

datalogger→get_autoStart()

Retourne le mode d'activation automatique de l'enregistreur de données à la mise sous tension.

cpp m pas vb cs java uwp py php ts es dnp cmd

datalogger→get_beaconDriven()

Retourne vrai si l'enregistreur de données est synchronisé avec la balise de localisation.

cpp m pas vb cs java uwp py php ts es dnp cmd

datalogger→get_currentRunIndex()

Retourne le numéro du Run actuel, correspondant au nombre de fois que le module a été mis sous tension avec la fonction d'enregistreur de données active.

cpp m pas vb cs java uwp py php ts es dnp cmd

datalogger→get_dataSets()

Retourne une liste d'objets `YDataSet` permettant de récupérer toutes les mesures stockées par l'enregistreur de données.

cpp m pas vb cs java uwp py php ts es dnp cmd

datalogger→get_dataStreams(v)

Construit une liste de toutes les séquences de mesures mémorisées par l'enregistreur (ancienne méthode).

datalogger→get_errorMessage()

Retourne le message correspondant à la dernière erreur survenue lors de l'utilisation de l'enregistreur de données.

cpp m pas vb cs java py php ts es

datalogger→get_errorType()

Retourne le code d'erreur correspondant à la dernière erreur survenue lors de l'utilisation de l'enregistreur de données.

cpp m pas vb cs java py php ts es

datalogger→get_friendlyName()

Retourne un identifiant global de l'enregistreur de données au format `NOM_MODULE.NOM_FONCTION`.

cpp m cs java py php ts es dnp

datalogger→get_functionDescriptor()

Retourne un identifiant unique de type `YFUN_DESCR` correspondant à la fonction.

cpp m pas vb cs java py php ts es

datalogger→get_functionId()

Retourne l'identifiant matériel de l'enregistreur de données, sans référence au module.

cpp m vb cs java py php ts es dnp

datalogger→get_hardwareId()

Retourne l'identifiant matériel unique de l'enregistreur de données au format `SERIAL.FUNCTIONID`.

cpp m vb cs java py php ts es dnp

datalogger→get_logicalName()

Retourne le nom logique de l'enregistreur de données.

cpp m pas vb cs java uwp py php ts es dnp cmd

datalogger→get_module()

Retourne l'objet `YModule` correspondant au module Yoctopuce qui héberge la fonction.

cpp m pas vb cs java py php ts es dnp

datalogger→get_module_async(callback, context)

Retourne l'objet `YModule` correspondant au module Yoctopuce qui héberge la fonction.

datalogger→get_recording()

Retourne l'état d'activation de l'enregistreur de données.

cpp m pas vb cs java uwp py php ts es dnp cmd

datalogger→get_serialNumber()

Retourne le numéro de série du module, préprogrammé en usine.

cpp m pas vb cs java uwp py php ts es dnp cmd

datalogger→get_timeUTC()

Retourne le timestamp Unix de l'heure UTC actuelle, lorsqu'elle est connue.

cpp m pas vb cs java uwp py php ts es dnp cmd

datalogger→get_usage()

Retourne le pourcentage d'utilisation de la mémoire d'enregistrement.

cpp m pas vb cs java uwp py php ts es dnp cmd

datalogger→get_userData()

Retourne le contenu de l'attribut userData, précédemment stocké à l'aide de la méthode `set_userData`.

cpp m pas vb cs java py php ts es

datalogger→isOnline()

Vérifie si le module hébergeant l'enregistreur de données est joignable, sans déclencher d'erreur.

cpp m pas vb cs java py php ts es dnp

datalogger→isOnline_async(callback, context)

Vérifie si le module hébergeant l'enregistreur de données est joignable, sans déclencher d'erreur.

datalogger→isReadOnly()

Indique si la fonction est en lecture seule.

cpp m pas vb cs java uwp py php ts es dnp cmd

datalogger→load(msValidity)

Met en cache les valeurs courantes de l'enregistreur de données, avec une durée de validité spécifiée.

cpp m pas vb cs java py php ts es

datalogger→loadAttribute(attrName)

Retourne la valeur actuelle d'un attribut spécifique de la fonction, sous forme de texte, le plus rapidement possible mais sans passer par le cache.

cpp m pas vb cs java uwp py php ts es dnp

datalogger→load_async(msValidity, callback, context)

Met en cache les valeurs courantes de l'enregistreur de données, avec une durée de validité spécifiée.

datalogger→muteValueCallbacks()

Désactive l'envoi de chaque changement de la valeur publiée au hub parent.

cpp m pas vb cs java uwp py php ts es dnp cmd

datalogger→nextDataLogger()

Continue l'énumération des enregistreurs de données commencée à l'aide de `yFirstDataLogger()` Attention, vous ne pouvez faire aucune supposition sur l'ordre dans lequel les enregistreurs de données sont retournés.

cpp m pas vb cs java uwp py php ts es

datalogger→registerValueCallback(callback)

Enregistre la fonction de callback qui est appelée à chaque changement de la valeur publiée.

cpp m pas vb cs java uwp py php ts es

datalogger→set_autoStart(newval)

Modifie le mode d'activation automatique de l'enregistreur de données à la mise sous tension.

cpp m pas vb cs java uwp py php ts es dnp cmd

datalogger→set_beaconDriven(newval)

Modifie le mode de synchronisation de l'enregistreur de données .

cpp m pas vb cs java uwp py php ts es dnp cmd

datalogger→set_logicalName(newval)

Modifie le nom logique de l'enregistreur de données.

cpp m pas vb cs java uwp py php ts es dnp cmd

datalogger→set_recording(newval)

Modifie l'état d'activation de l'enregistreur de données.

cpp m pas vb cs java uwp py php ts es dnp cmd

datalogger→set_timeUTC(newval)

Modifie la référence de temps UTC, afin de l'attacher aux données enregistrées.

cpp m pas vb cs java uwp py php ts es dnp cmd

datalogger→set_userData(data)

Enregistre un contexte libre dans l'attribut userData de la fonction, afin de le retrouver plus tard à l'aide de la méthode `get_userData`.

cpp m pas vb cs java py php ts es

datalogger→unmuteValueCallbacks()

Réactive l'envoi de chaque changement de la valeur publiée au hub parent.

cpp m pas vb cs java uwp py php ts es dnp cmd

datalogger→wait_async(callback, context)

Attend que toutes les commandes asynchrones en cours d'exécution sur le module soient terminées, et appelle le callback passé en paramètre.

ts es

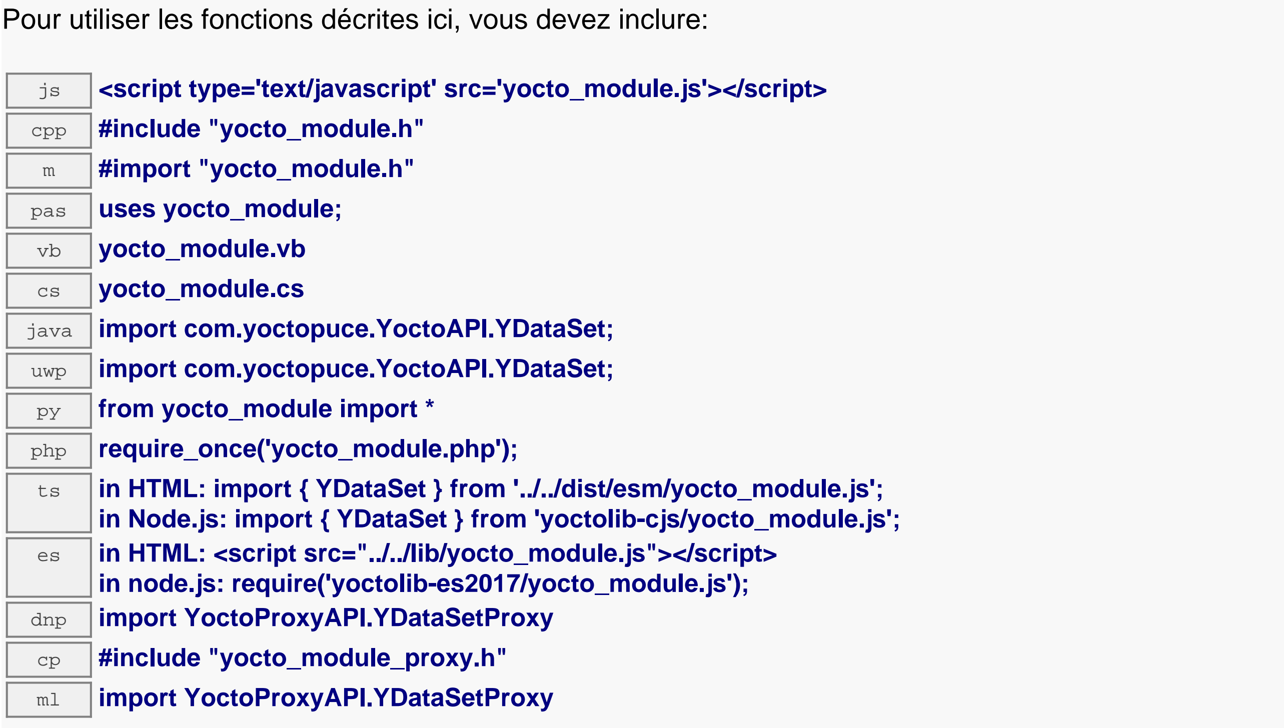
25.8. La classe YDataSet

Séquence de données enregistrées par le datalogger, obtenue par la méthode `sensor.get_recordedData()`

Les objets `YDataSet` permettent de récupérer un ensemble de mesures enregistrées correspondant à un capteur donné, pour une période choisie. Ils permettent le chargement progressif des données. Lorsque l'objet `YDataSet` est instancié par la méthode `sensor.get_recordedData()`, aucune donnée n'est encore chargée du module. Ce sont les appels successifs à la méthode `loadMore()` qui procèdent au chargement effectif des données depuis l'enregistreur de données.

Un résumé des mesures disponibles est disponible via la fonction `get_preview()` dès le premier appel à `loadMore()`. Les mesures elles-même sont disponibles via la fonction `get_measures()` au fur et à mesure de leur chargement.

Cette classe ne fonctionne que si le module utilise un firmware relativement récent, car les objets `YDataSet` ne sont pas supportés par les firmwares antérieurs à la révision 13000.



Pour utiliser les fonctions décrites ici, vous devez inclure:

js	<script type='text/javascript' src='yocto_module.js'></script>
cpp	#include "yocto_module.h"
m	#import "yocto_module.h"
pas	uses yocto_module;
vb	yocto_module.vb
cs	yocto_module.cs
java	import com.yoctopuce.YoctoAPI.YDataSet;
uwp	import com.yoctopuce.YoctoAPI.YDataSet;
py	from yocto_module import *
php	require_once('yocto_module.php');
ts	in HTML: import { YDataSet } from '../../dist/esm/yocto_module.js'; in Node.js: import { YDataSet } from 'yoctolib-cjs/yocto_module.js';
es	in HTML: <script src="../../lib/yocto_module.js"></script> in node.js: require('yoctolib-es2017/yocto_module.js');
dnp	import YoctoProxyAPI.YDataSetProxy
cp	#include "yocto_module_proxy.h"
ml	import YoctoProxyAPI.YDataSetProxy

Fonction globales

YDataSet.Init(sensorName, startTime, endTime)

Retourne un objet `YDataSet` permettant de charger les mesures d'un capteur donné par son nom ou identifiant matériel, pour un intervalle de temps spécifié.

Méthodes des objets `YDataSet`

dataset→get_endTimeUTC()

Retourne l'heure absolue de la fin des mesures disponibles, sous forme du nombre de secondes depuis le 1er janvier 1970 (date/heure au format Unix).

cpp m pas vb cs java uwp py php ts es dnp

dataset→get_functionId()

Retourne l'identifiant matériel de la fonction qui a effectué les mesures, sans référence au module.

cpp m pas vb cs java uwp py php ts es dnp

dataset→get_hardwareId()

Retourne l'identifiant matériel unique de la fonction qui a effectué les mesures, au format SERIAL.FUNCTIONID.

cpp m pas vb cs java uwp py php ts es dnp

dataset→get_measures()

Retourne toutes les mesures déjà disponibles pour le DataSet, sous forme d'une liste d'objets YMeasure.

cpp m pas vb cs java uwp py php ts es dnp

dataset→get_measuresAt(measure)

Retourne les mesures détaillées pour une mesure résumée précédemment retournée par get_preview().

cpp m pas vb cs java uwp py php ts es dnp

dataset→get_measuresAvgAt(index)

Retourne la valeur moyenne observée durant l'intervalle de temps couvert par l'entrée spécifiée du résumé des mesures.

dataset→get_measuresEndTimeAt(index)

Retourne l'heure de fin de l'entrée spécifiée du résumé des mesures, sous forme du nombre de secondes depuis le 1er janvier 1970 UTC (date/heure au format Unix).

dataset→get_measuresMaxAt(index)

Retourne la plus grande valeur observée durant l'intervalle de temps couvert par l'entrée spécifiée du résumé des mesures.

dataset→get_measuresMinAt(index)

Retourne la plus petite valeur observée durant l'intervalle de temps couvert par l'entrée spécifiée du résumé des mesures.

dataset→get_measuresRecordCount()

Retourne le nombre de mesures déja chargées pour ce DataSet.

dataset→get_measuresStartTimeAt(index)

Retourne l'heure absolue de l'entrée spécifiée du résumé des mesures, sous forme du nombre de secondes depuis le 1er janvier 1970 UTC (date/heure au format Unix).

dataset→get_preview()

Retourne une version résumée des mesures qui pourront être obtenues de ce YDataSet, sous forme d'une liste d'objets YMeasure.

cpp m pas vb cs java uwp py php ts es dnp

dataset→get_previewAvgAt(index)

Retourne la valeur moyenne observée durant l'intervalle de temps couvert par l'entrée spécifiée du résumé des mesures.

dataset→get_previewEndTimeAt(index)

Retourne l'heure de fin de l'entrée spécifiée du résumé des mesures, sous forme du nombre de secondes depuis le 1er janvier 1970 UTC (date/heure au format Unix).

dataset→get_previewMaxAt(index)

Retourne la plus grande valeur observée durant l'intervalle de temps couvert par l'entrée spécifiée du résumé des mesures.

dataset→get_previewMinAt(index)

Retourne la plus petite valeur observée durant l'intervalle de temps couvert par l'entrée spécifiée du résumé des mesures.

dataset→get_previewRecordCount()

Retourne le nombre d'entrées dans la version résumée des mesures qui pourront être obtenues dans ce YDataSet.

dataset→get_previewStartTimeAt(index)

Retourne l'heure absolue de l'entrée spécifiée du résumé des mesures, sous forme du nombre de secondes depuis le 1er janvier 1970 UTC (date/heure au format Unix).

dataset→get_progress()

Retourne l'état d'avancement du chargement des données, sur une échelle de 0 à 100.

cpp m pas vb cs java uwp py php ts es dnp

dataset→get_startTimeUTC()

Retourne l'heure absolue du début des mesures disponibels, sous forme du nombre de secondes depuis le 1er janvier 1970 (date/heure au format Unix).

cpp m pas vb cs java uwp py php ts es dnp

dataset→get_summary()

Retourne un objet `YMeasure` résumant tout le `YDataSet`.

cpp m pas vb cs java uwp py php ts es dnp

dataset→get_summaryAvg()

Retourne la valeur moyenne observée durant l'intervalle de temps couvert par ce `DataSet`.

dataset→get_summaryEndTime()

Retourne l'heure de fin de la dernière mesure du data set, sous forme du nombre de secondes depuis le 1er janvier 1970 UTC (date/heure au format Unix).

dataset→get_summaryMax()

Retourne la plus grande valeur observée durant l'intervalle de temps couvert par ce `DataSet`.

dataset→get_summaryMin()

Retourne la plus petite valeur observée durant l'intervalle de temps couvert par ce `DataSet`.

dataset→get_summaryStartTime()

Retourne l'heure absolue de la première mesure du data set, sous forme du nombre de secondes depuis le 1er janvier 1970 UTC (date/heure au format Unix).

dataset→get_unit()

Retourne l'unité dans laquelle la valeur mesurée est exprimée.

cpp m pas vb cs java uwp py php ts es dnp

dataset→loadMore()

Procède au chargement du bloc suivant de mesures depuis l'enregistreur de données du module, et met à jour l'indicateur d'avancement.

cpp m pas vb cs java uwp py php ts es dnp

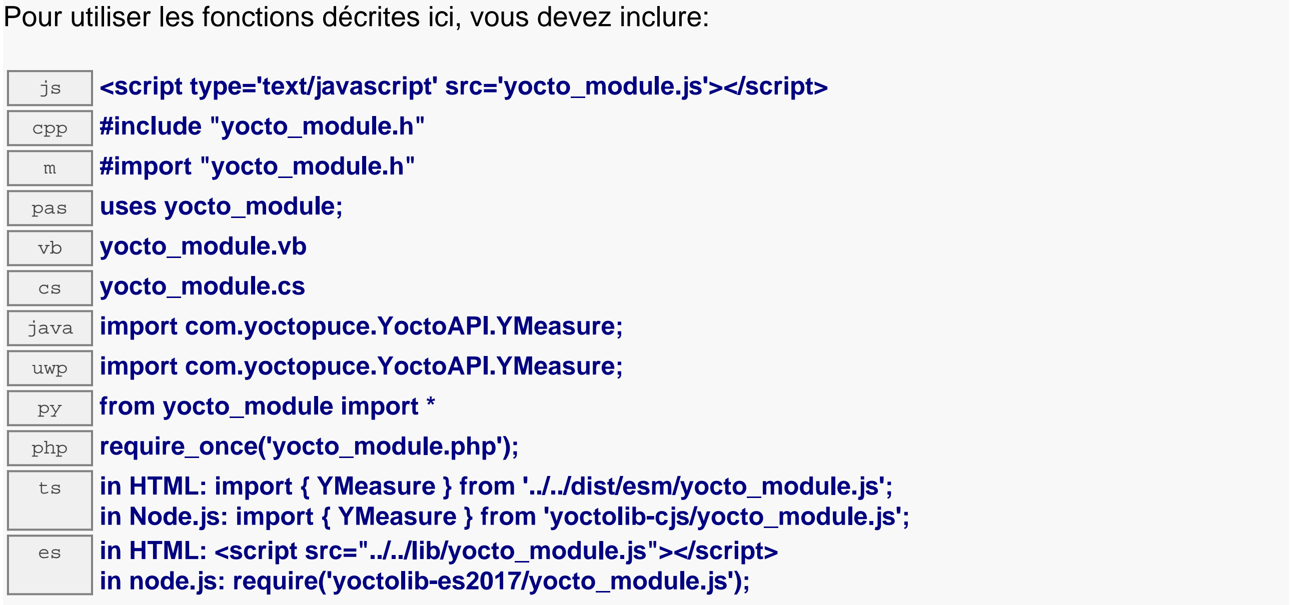
dataset→loadMore_async(callback, context)

Procède au chargement du bloc suivant de mesures depuis l'enregistreur de données du module, de manière asynchrone.

25.9. La classe YMeasure

Valeur mesurée, retournée en particulier par les méthodes de la classe `YDataSet`.

Les objets `YMeasure` sont utilisés dans l'interface de programmation Yoctopuce pour représenter une valeur observée à un moment donnée. Ces objets sont utilisés en particulier en conjonction avec la classe `YDataSet`, mais aussi par les notification périodique des capteurs configurées (voir `sensor.registerTimedReportCallback`).



Pour utiliser les fonctions décrites ici, vous devez inclure:

js	**<script type='text/javascript' src='yocto_module.js'></script>**
cpp	**#include "yocto_module.h"**
m	**#import "yocto_module.h"**
pas	**uses yocto_module;**
vb	**yocto_module.vb**
cs	**yocto_module.cs**
java	**import com.yoctopuce.YoctoAPI.YMeasure;**
uwp	**import com.yoctopuce.YoctoAPI.YMeasure;**
py	**from yocto_module import ***
php	**require_once('yocto_module.php');**
ts	**in HTML: import { YMeasure } from '../../dist/esm/yocto_module.js';** **in Node.js: import { YMeasure } from 'yoctolib-cjs/yocto_module.js';**
es	**in HTML: <script src="../../lib/yocto_module.js"></script>** **in node.js: require('yoctolib-es2017/yocto_module.js');**

Méthodes des objets `YMeasure`

measure→get_averageValue()
Retourne la valeur moyenne observée durant l'intervalle de temps couvert par la mesure.
cpp m pas vb cs java uwp py php ts es

measure→get_endTimeUTC()
Retourne l'heure absolue de la fin de la mesure, sous forme du nombre de secondes depuis le 1er janvier 1970 UTC (date/heure au format Unix).
cpp m pas vb cs java uwp py php ts es

measure→get_maxValue()
Retourne la plus grande valeur observée durant l'intervalle de temps couvert par la mesure.
cpp m pas vb cs java uwp py php ts es

measure→get_minValue()
Retourne la plus petite valeur observée durant l'intervalle de temps couvert par la mesure.
cpp m pas vb cs java uwp py php ts es

measure→get_startTimeUTC()
Retourne l'heure absolue du début de la mesure, sous forme du nombre de secondes depuis le 1er janvier 1970 UTC (date/heure au format Unix).
cpp m pas vb cs java uwp py php ts es

26. Problèmes courants

26.1. Par où commencer ?

Si c'est la première fois que vous utilisez un module Yoctopuce et ne savez pas trop par où commencer, allez donc jeter un coup d'œil sur le blog de Yoctopuce. Il y a une section dédiée aux débutants [1].

26.2. Linux et USB

Pour fonctionner correctement sous Linux, la librairie a besoin d'avoir accès en écriture à tous les périphériques USB Yoctopuce. Or, par défaut, sous Linux les droits d'accès des utilisateurs non-root à USB sont limités à la lecture. Afin d'éviter de devoir lancer les exécutables en tant que root, il faut créer une nouvelle règle *udev* pour autoriser un ou plusieurs utilisateurs à accéder en écriture aux périphériques Yoctopuce.

Pour ajouter une règle *udev* à votre installation, il faut ajouter un fichier avec un nom au format "`##-nomArbitraire.rules`" dans le répertoire "`/etc/udev/rules.d`". Lors du démarrage du système, *udev* va lire tous les fichiers avec l'extension "`.rules`" de ce répertoire en respectant l'ordre alphabétique (par exemple, le fichier "`51-custom.rules`" sera interprété APRES le fichier "`50-udev-default.rules`").

Le fichier "`50-udev-default`" contient les règles *udev* par défaut du système. Pour modifier le comportement par défaut du système, il faut donc créer un fichier qui commence par un nombre plus grand que 50, qui définira un comportement plus spécifique que le défaut du système. Notez que pour ajouter une règle vous aurez besoin d'avoir un accès root sur le système.

Dans le répertoire `udev_conf` de l'archive de VirtualHub[2] pour Linux, vous trouverez deux exemples de règles qui vous éviteront de devoir partir de rien.

Exemple 1: 51-yoctopuce.rules

Cette règle va autoriser tous les utilisateurs à accéder en lecture et en écriture aux périphériques Yoctopuce USB. Les droits d'accès pour tous les autres périphériques ne seront pas modifiés. Si ce scénario vous convient, il suffit de copier le fichier "`51-yoctopuce_all.rules`" dans le répertoire "`/etc/udev/rules.d`" et de redémarrer votre système.

[1] voir: http://www.yoctopuce.com/FR/blog_by_categories/pour-les-debutants
[2] http://www.yoctopuce.com/EN/virtualhub.php

```
# udev rules to allow write access to all users
# for Yoctopuce USB devices
SUBSYSTEM=="usb", ATTR{idVendor}=="24e0", MODE="0666"
```

Exemple 2: 51-yoctopuce_group.rules

Cette règle va autoriser le groupe "yoctogroup" à accéder en lecture et écriture aux périphériques Yoctopuce USB. Les droits d'accès pour tous les autres périphériques ne seront pas modifiés. Si ce scénario vous convient, il suffit de copier le fichier "`51-yoctopuce_group.rules`" dans le répertoire "`/etc/udev/rules.d`" et de redémarrer votre système.

```
# udev rules to allow write access to all users of "yoctogroup"
# for Yoctopuce USB devices
SUBSYSTEM=="usb", ATTR{idVendor}=="24e0", MODE="0664", GROUP="yoctogroup"
```

26.3. Plateformes ARM: HF et EL

Sur ARM il existe deux grandes familles d'executables: HF (Hard Float) et EL (EABI Little Endian). Ces deux familles ne sont absolument pas compatibles entre elles. La capacité d'une machine ARM à faire tourner des exécutables de l'une ou l'autre de ces familles dépend du hardware et du système d'exploitation. Les problèmes de compatibilité entre ArmHL et ArmEL sont assez difficiles à diagnostiquer, souvent même l'OS se révèle incapable de distinguer un exécutable HF d'un exécutable EL.

Tous les binaires Yoctopuce pour ARM sont fournis pré-compilée pour ArmHF et ArmEL, si vous ne savez à quelle famille votre machine ARM apartient, essayez simplement de lancer un exécutable de chaque famille.

26.4. Les exemples de programmation n'ont pas l'air de marcher

La plupart des exemples de programmation de l'API Yoctopuce sont des programmes en ligne de commande et ont besoin de quelques paramètres pour fonctionner. Vous devez les lancer depuis l'invite de commande de votre système d'exploitation ou configurer votre IDE pour qu'il passe les paramètres corrects au programme [3].

26.5. Module alimenté mais invisible pour l'OS

Si votre Yocto-SDI12 est branché par USB et que sa LED bleue s'allume, mais que le module n'est pas vu par le système d'exploitation, vérifiez que vous utilisez bien un vrai câble USB avec les fils pour les données, et non pas un câble de charge. Les câbles de charge n'ont que les fils d'alimentation.

26.6. Another process named xxx is already using yAPI

Si lors de l'initialisation de l'API Yoctopuce, vous obtenez le message d'erreur "*Another process named xxx is already using yAPI*", cela signifie qu'une autre application est déjà en train d'utiliser les modules Yoctopuce USB. Sur une même machine, un seul processus à la fois peut accéder aux modules Yoctopuce par USB. Cette limitation peut facilement être contournée en utilisant un VirtualHub et le mode réseau [4].

[3] voir: http://www.yoctopuce.com/FR/article/a-propos-des-programmes-d-exemples
[4] voir: http://www.yoctopuce.com/FR/article/message-d-erreur-another-process-is-already-using-yapi

26.7. Déconnexions, comportement erratique

Si votre Yocto-SDI12 se comporte de manière erratique et/ou se déconnecte du bus USB sans raison apparente, vérifiez qu'il est alimenté correctement. Evitez les câbles d'une longueur supérieure à 2 mètres. Au besoin, intercalez un hub USB alimenté [5] [6].

26.8. Le module ne marche plus après une mise à jour ratée

Si une mise à jour du firmware de votre Yocto-SDI12 échoue, il est possible que le module ne soit plus en état de fonctionner. Si c'est le cas, branchez votre module en maintenant sont Yocto-Bouton pressé. La Yocto-LED devrait s'allumer en haute intensité et rester fixe. Relâchez le bouton. Votre Yocto-SDI12 devrait alors apparaître dans le bas de l'interface du virtualHub comme un module attendant une mise à jour de firmware. Cette mise à jour aura aussi pour effet de réinitialiser le module à sa configuration d'usine.

26.9. RegisterHub d'une instance de VirtualHub déconnecte la précédente

Si lorsque vous faire un YAPI.RegisterHub de VirtualHub et que la connexion avec un autre VirtualHub précédement enregistré tombe, vérifiez que les machines qui hébergent ces VirtualHubs ont bien un *hostname* différent. Ce cas de figure est très courant avec les machines dont le système d'exploitation est installé avec une image monolithique, comme les Raspberry Pi par exemple. L'API Yoctopuce utilise les numéros de série Yoctopuce pour communiquer et le numéro de série d'un VirtualHub est créé à la volée à partir du *hostname* de la machine qui l'héberge.

26.10. Commandes ignorées

Si vous avez l'impression que des commandes envoyées à un module Yoctopuce sont ignorées, typiquement lorsque vous avez écrit un programme qui sert à configurer ce modules Yoctopuce et qui envoie donc beaucoup de commandes, vérifiez que vous avez bien mis un YAPI.FreeAPI() à la fin du programme. Les commandes sont envoyées aux modules de manière asynchrone grâce à un processus qui tourne en arrière plan. Lorsque le programme se termine, ce processus est tué, même s'il n'a pas eu le temps de tout envoyer. En revanche API.FreeAPI() attend que la file d'attente des commandes à envoyer soit vide avant de libérer les ressources utilisées par l'API et rendre la main.

26.11. Module endommagé

Yoctopuce s'efforce de réduire la production de déchets électroniques. Si vous avez l'impression que votre Yocto-SDI12 ne fonctionne plus, commencez par contacter le support Yoctopuce par e-mail pour poser un diagnostic. Même si c'est suite à une mauvaise manipulation que le module a été endommagé, il se peut que Yoctopuce puisse le réparer, et ainsi éviter de créer un déchet électronique.

Déchets d'équipements électriques et électroniques (DEEE) Si voulez vraiment vous débarasser de votre Yocto-SDI12, ne le jetez pas à la poubelle, mais ramenez-le à l'un des points de collecte proposé dans votre région afin qu'il soit envoyé à un centre de recyclage ou de traitement spécialisé.

[5] voir: http://www.yoctopuce.com/FR/article/cables-usb-la-taille-compte
[6] voir: http://www.yoctopuce.com/FR/article/combien-de-capteurs-usb-peut-on-connecter

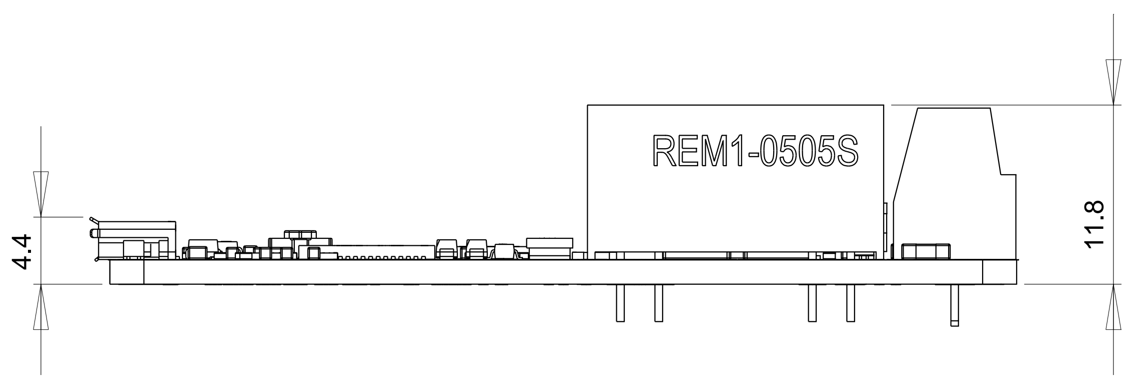
27. Caractéristiques

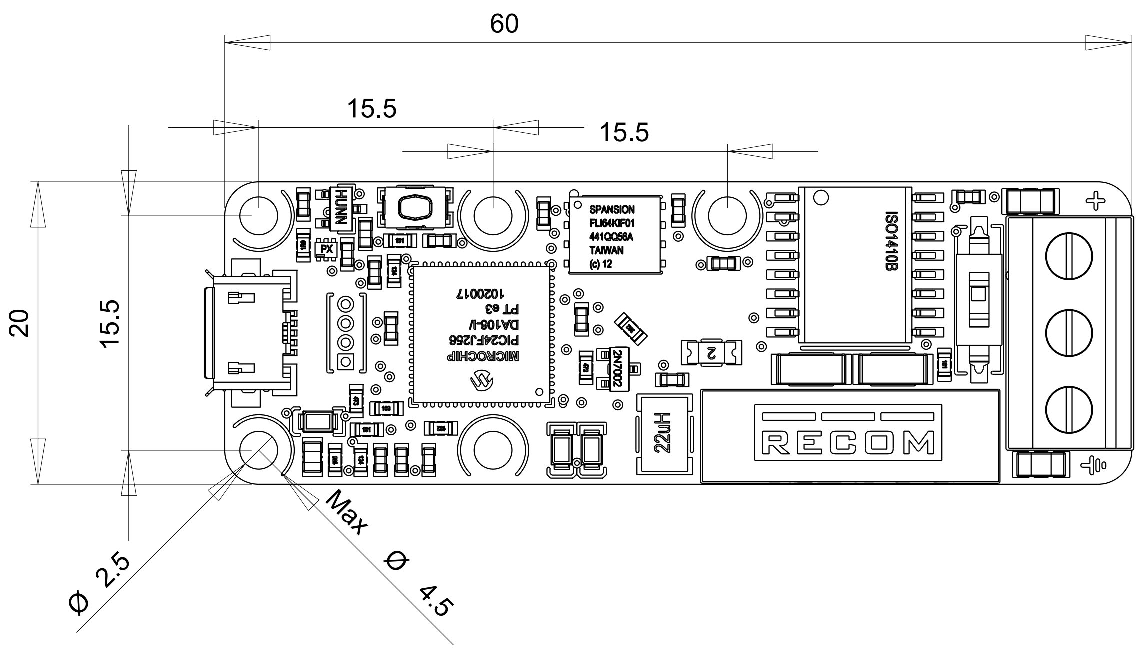
Vous trouverez résumées ci-dessous les principales caractéristiques techniques de votre module Yocto-SDI12

Identifiant produit	YSDIMK01
Révision matérielle†	
Connecteur USB	micro-B
Largeur	20 mm
Longueur	60 mm
Poids	11 g
Classe de protection selon IEC 61140	classe III
Temp. de fonctionnement normale	5...40 °C
Temp. de fonctionnement étendue‡	-30...85 °C
Conformité RoHS	RoHS III (2011/65/UE+2015/863)
USB Vendor ID	0x24E0
USB Device ID	0x00AB
Boîter recommandé	YoctoBox-Long-Thick-Black
Code tarifaire harmonisé	9032.9000
Fabriqué en	Suisse

† Ces spécifications correspondent à la révision matérielle actuelle du produit. Les spécifications des versions antérieures peuvent être inférieures.

‡ La plage de température étendue est définie d'après les spécifications des composants et testée sur une durée limitée (1h). En cas d'utilisation prolongée hors de la plage de température standard, il est recommandé procéder à des tests extensifs avant la mise en production.





All dimensions are in mm
Toutes les dimensions sont en mm

Yocto-RS485-V2

A4

Scale
2:1
Echelle