Yoctopuce Toolbox for MATLAB

(version Beta)

This library implements a set of classes enabling you to use Yoctopuce sensors and actuators within MATLAB. It includes a dynamic library written in C++ to take care of low level communications with the modules without requiring extra device drivers.

First steps

All the classes are assembled in a package named **YoctoProxyAPI**. To simplify the code below, we import all the classes of the package, but it is obviously not compulsory:

```
import YoctoProxyAPI.*
```

All the general functions of the library are gathered in the `YAPIProxy` class in the shape of static methods. The first method that you must call, **`RegisterHub()`**, is used to indicate where the Yoctopuce modules are located. The library can use modules connected through the network or directly connected by USB.

```
YAPIProxy.RegisterHub('usb')
```

```
ans =

  0×0 empty char array
```

If we had wanted to use modules connected via a YoctoHub-Wireless-n for example, we would have replaced the **`'usb'`** string with an IP address such as **`'192.168.1.100'`**.

Let's now see how to list all the detected modules:

```
YModuleProxy.GetSimilarFunctions()
```

```
ans = 2×1 cell
'THRMCPL1-13405C.module'
'MXPWRRLY-FA815.module'
```

This function returns a list of hardware identifiers, corresponding to each module that was found. In a similar way, we can list all the temperature sensors, whatever their type or their distribution on physical modules, in the case of modules with several inputs:

```
YTemperatureProxy.GetSimilarFunctions()
```

```
ans = 2×1 cell
'THRMCPL1-13405C.temperature2'
'THRMCPL1-13405C.temperature1'
```

Driving command modules

The Yoctopuce library provides distinct classes enabling you to drive the outputs of command modules through properties. For example, with the **`YRelayProxy`** class, you can drive relay outputs, and with the **`YCurrentLoopOutputProxy`** class you can drive outputs in 4-20mA current loop. Here is a first example of use:

```
pump = YRelayProxy.FindRelay('MXPWRRLY-FA815.relay3');
pump.State = 'State_B';
```

If you have given a logical name to the output, for example with the VirtualHub, you can also find the relay through its name:

```
pump = YRelayProxy.FindRelay('WaterPump');
pump.pulse(2000);
```

In this second example, rather than commuting the relay in a definitive manner, we requested it to switch to the B state for a 2000ms duration only.

You can easily consult the interesting properties of the interface objects:

```
pump
```

```
pump =
  YoctoProxyAPI.YRelayProxy with properties:

   General
    TargetFunction: 'WaterPump'
        HardwareId: 'MXPWRRLY-FA815.relay3'
      FriendlyName: 'MXPWRRLY-FA815.WaterPump'
             State: State_A

  Show all properties
```

Using sensor type modules

To use sensors, the principle is very similar: you look for the wanted function through its hardware name or the logical name that you have previously assigned to it, then you can read its value in the **`CurrentValue`** property. The measuring unit is available in the **`Unit`** property. If you display the object, you can find other useful properties of the sensor, such as here in the case of a Yocto-Thermocouple:

```
tc = YTemperatureProxy.FindTemperature('')
```

```
tc =
  YoctoProxyAPI.YTemperatureProxy with properties:

   General
    TargetFunction: '(any)'
        HardwareId: 'THRMCPL1-13405C.temperature2'
      FriendlyName: 'THRMCPL1-13405C.temperature2'
      CurrentValue: 27.6200
              Unit: ''C'

  Show all properties
```

```
measure = sprintf("%f %s", tc.CurrentValue, tc.Unit)
```

```
measure =
"27.620000 'C"
```

You can note that the description of the object includes not only the measured temperature, but also the kind of thermocouple which has been configured.

A complete example

Here is a short example of a control process:

```
import YoctoProxyAPI.*
YAPIProxy.RegisterHub('usb');
light = YLightSensorProxy.FindLightSensor('diningRoom');
lightControl = YRelayProxy.FindRelay('lightSwitch');
redButton = YAnButtonProxy.FindAnButton('redButton');

while ~redButton.IsPressed
    if light.CurrentValue < 30
        % turn on the light
        lightControl.State = 'State_B';
    end
end

% Frees all the resources
YAPIProxy.FreeAPI();
```

A few comments on the code:

- As you can see, most of the interaction with the Yoctopuce modules is performed through the properties, which makes reading the code easier.
- Moreover, we guaranty that access to properties is almost instantaneous, as the properties are refreshed in a background task through an independent process. It's the specificity of our "Proxy" libraries.
- The call to **`FreeAPI()`** at the end frees the USB port and disconnect the MATLAB objects from the Yoctopuce modules.

Supported platforms

All the classes are defined in MATLAB language, but they delegate a large amount of the work to the C++ Proxy library, provided as a dynamic library. The toolbox distribution includes the dynamic library binaries for Windows, and Linux in 32/64 bits, and for MacOS in 64 bits, which should enable you to use it on all these platforms (we haven't tested MacOS yet). The source code of the dynamic library is available.

What is not yet available in this Beta version

Examples for each Yoctopuce module

All the Yoctopuce libraries include an example of use for each module. The MATLAB library won't make exception, you're entitled to a Live Script for each module when the library is officially available. In the mean time, you'll have to make do with the examples in this file. Online help in MATLAB should however enable you to find what you need during the Beta-test period.

Exhaustive documentation

We haven't yet generated the reference file detailing the use of each class. However, you already have the online help in MATLAB. If you want explanations on the properties and the list of classes, you can meanwhile read the documentation of the ".NET Proxy" library which is based on the same structure.

Use of the data logger

Yoctopuce sensors are able to record measures on their embedded flash memory. We have planned to allow easy access to it as *TimeTables*, but this isn't yet available. The only problem with *Time Tables* is that they appeared only in MATLAB 2016b. If possible, we will therefore provide an interface based on *Time Series* as well.

Use with Simulink

We have decided to implement our library classes as MATLAB System Objects. This should enable you to use them directly in Simulink with System Blocks, but we will come back to this topic later when we will have made more headway in this matter.