
Wyliodrin Studio Documentation

Release 2.2.3-beta

Diana Ghindaoanu

May 30, 2021

Contents

1	Getting Started	5
2	Boards Setup	9
3	General Architecture of Wylodrin STUDIO	43
4	Extension methods	53
5	Deploy Application	61
6	Wylodrin Studio API	67
7	How to write a plugin	97
8	Translations	115
9	Dialogs and Notifications	119
10	Emulators	127
11	Simulators	131
	Index	143

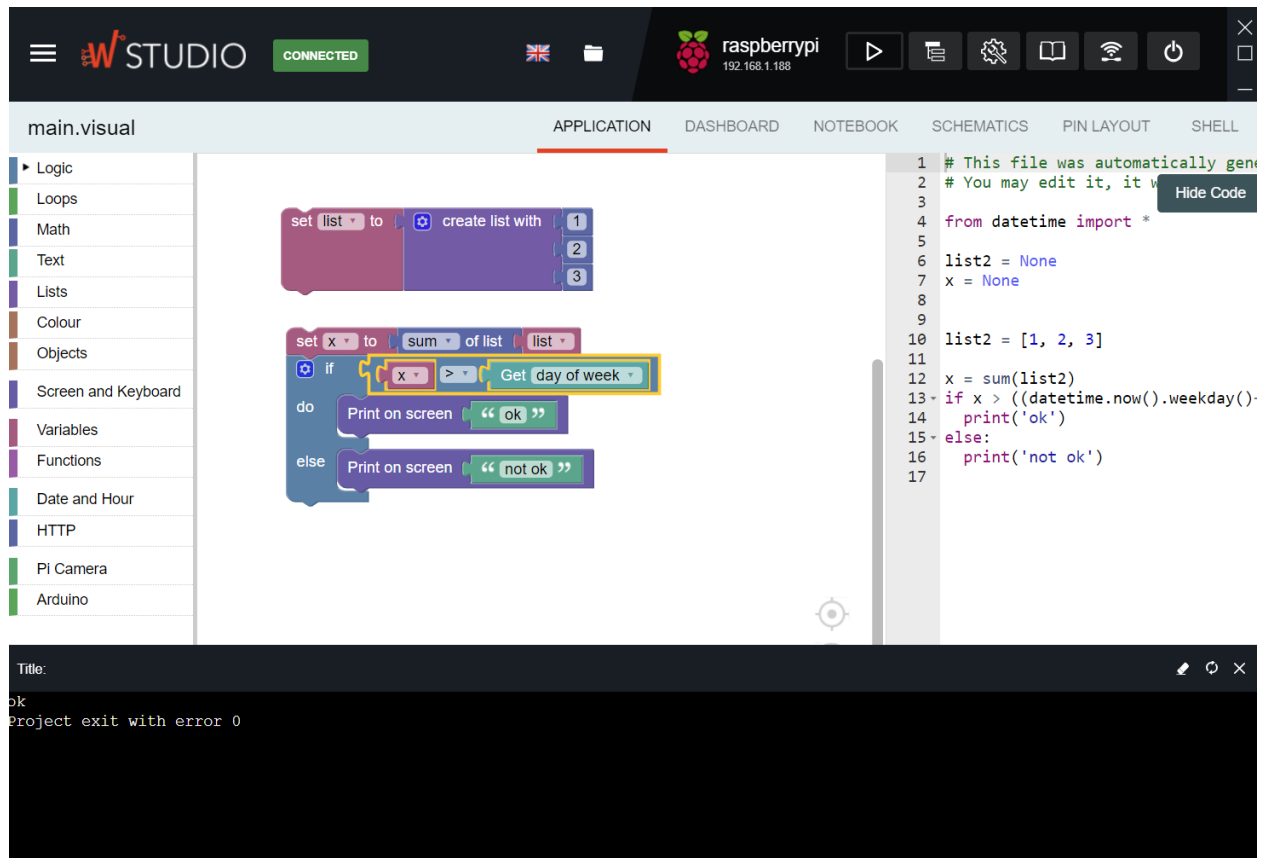
Wylidrin STUDIO is an educational platform for software and hardware development for IoT and Embedded Linux systems.

The application has been built as an extendable framework. The main architecture is a collection of plugins that add functionality. This approach has been chosen as different devices have very different ways of connecting and interacting with the computer and/or the browser.

The plugin system allows Wylidrin STUDIO to be very flexible and extendable. Adding features such as supported devices or languages and even very different new functionalities is a matter of writing a new plugin.

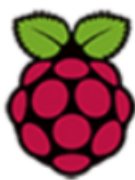
The purpose of Wylidrin STUDIO is to help its users deploy industrial IoT application, gain IoT knowledge and offer customized results in the same domain, by providing them a series of professional solutions:

- Connect to devices using TCP/IP or serial port
- Develop software and firmware for IoT in several programming languages
- Shell access to the device
- Import and export Wylidrin STUDIO projects
- Visual dashboard for displaying sensor data
- Display the hardware schematics
- Manage packages for Python and Javascript
- Task manager for managing the device
- Network connection manager for the device (Ethernet and WiFi)
- Interactive electronics documentation (resistor color code)
- Example projects and firmware
- Wylidrin API documentation in C/C++, Python and Javascript



For the moment, the devices supported by the platform are:

- Raspberry Pi
- MicroPython
- UDOO Neo
- BeagleBone Black



Also, the recognized programming languages at the time are:

- Javascript
- Python
- C/C++
- Rust
- Shell Script (bash)

- Visual Programming (translates to Python)

JavaScript



Python



Bash Shell



Visual



Wyliodrin STUDIO is available in two versions: an offline or downloadable one and a web version.

1.1 Download the application

For **Windows** users:

Wyliodrin STUDIO beta_Windows 64 bit

For **Linux** users:

Wyliodrin STUDIO beta_Linux 64 bit

For **Mac OS** users:

Wyliodrin STUDIO beta_macOS

1.2 Use the web version

You also have the possibility to run and use a browser version of Wylidrin Studio, by copying the following link into your browser address bar:

beta.wylidrin.studio

1.3 Build from source

If you wish to contribute to the improvement of the application or if you want to add your own features or plugins, our code is *open source*, which means you can clone it from our Github.

To download the source code, you must have a GitHub account. Open a terminal, choose the folder where you want to clone our repository and run the following command:

```
git clone https://github.com/wylidrinstudio/WylidrinSTUDIO
```

There are 2 methods to build and run the application:

To build the **STANDALONE** version, you will have to run the following commands:

```
npm install
npx electron-rebuild
npx webpack
npm run electron
```

To run the **STANDALONE** version , you will have to run the following command:

```
npm start
```

To build **BROWSER** version, you will have to delete the *build* folder, run:

```
npm install
npx webpack --config=webpack.browser.config.js
cd build
npm install
```

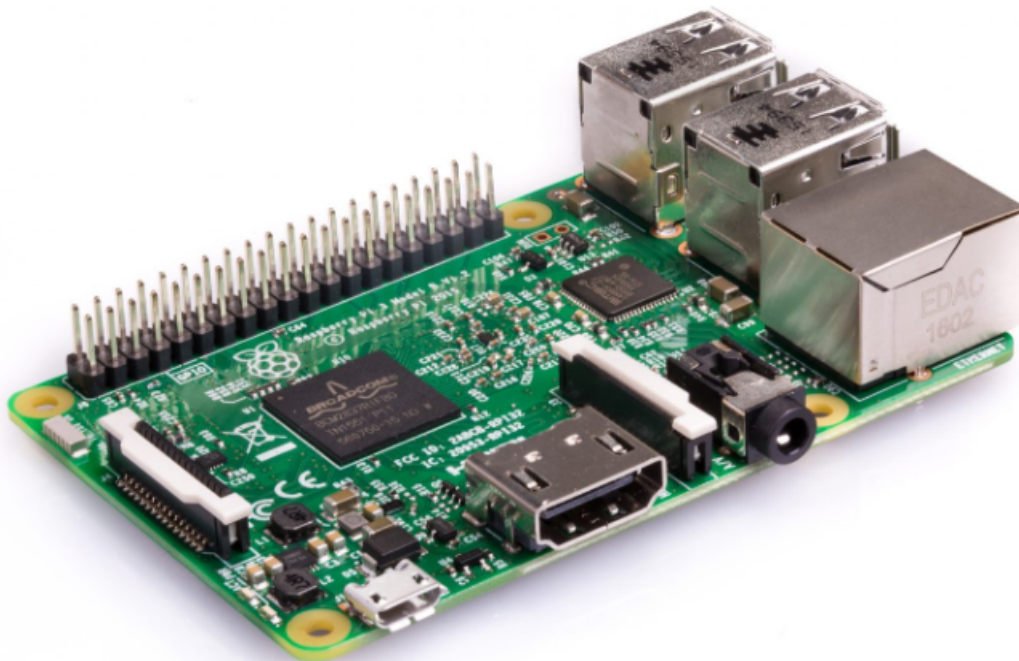
To run the **BROWSER** version , you will have to run the following command:

```
npm start
```

Once the application was installed and built, you can make changes on our source code, in order to improve it.

2.1 Raspberry Pi

This will show how to set up a Raspberry Pi device.



2.1.1 Video

A quick video explaining how to connect the Raspberry Pi. A detailed tutorial is available below.

2.1.2 Download the pre-configured image

The easiest way to set up a Raspberry Pi board so that it becomes available for Wylodrin STUDIO is to download an image that is already configured.

Download the image for [Raspberry Pi Zero and Raspberry Pi 1](#).

Download the image for [Raspberry Pi 2](#), [Raspberry Pi 3](#) and [Raspberry Pi 4](#).

Once the image downloaded and unzipped, the only thing that you have to do is to *flash* it. After that, you can simply insert the SD card into the Raspberry Pi and your board should be visible within Wylodrin STUDIO.

2.1.3 Set up the board manually

However, you can also choose to configure the required image by yourself.

This will imply flashing an image with the OS (Raspbian), installing the STUDIO Supervisor container and setting up some configuration files.

Download the Raspbian image

You will need to:

1. Download a Raspberry Pi Image
2. Install the Studio Supervisor
3. Setup a provisioning file

Raspbian is provided in two flavors, **Desktop** and **Lite**. The first one is packed with all the desktop user interface and applications while the second one contains only the minimum OS without any applications. The second one is just what we want for Studio, as we will deploy all the applications that want.

Download the [Raspbian Lite](#) image from the Raspberry Pi foundation. This is the standard OS for the Raspberry Pi provided by the manufacturer.

Flash the image

The downloaded image needs to be flash (written) to an SD card. The minimum size of the SD card is 4 GB.

Note: We recommend a minimum of 8 GB Class 10 SD Card. For small applications 4 GB might be enough.

To flash the image, you will need a special software. The recommended application is [Etcher](#).

Note: For Linux users, you may use the **dd** utility.

Install STUDIO Supervisor

To be able to access the Studio network, the Raspberry Pi needs to run the STUDIO Supervisor software. The following tutorial will explain how to install it.

After writing the SD Card, insert it into the Raspberry Pi and start the Raspberry Pi. You will have to access it. This can be done either by:

- connecting the Raspberry Pi to the network and use a SSH to connect to it (If you are using **Raspberry Pi Zero** and you want to use SSH, you will need an USB-OTG adapter to get connected to the network.)
- connect a monitor and a keyboard to the Raspberry Pi

Note: Using the SSH will require to enable it before. Insert the SD card into your computer. One (or two if Linux) partitions will show up. On the FAT partition (the first one), create an empty file named **ssh**.

Install Dependencies

The dependencies you will have to install are:

- *supervisor*: allows you to monitor processes related to a project
- *redis*: database management system
- *build-essential*: reference package for all the packages required for compilation
- *git*: required for the **npm install** command to download git included package
- *python3-pip*: python 3 programming language
- *docker*: containerization technology

```
sudo apt-get update
sudo apt-get install -y supervisor redis build-essential git python3-pip docker-ce_
↪ docker-ce-cli containerd.io
```

(continues on next page)

(continued from previous page)

```
# To enable the Notebook tab, you should also run
sudo pip3 install redis pygments
```

Note: If the docker feature does not work, you can install it manually following the steps that can be found in **Install Docker manually**

Install Node.js

The next step is to [install NodeJS](#), considering the model of Raspberry Pi that you are using.

For **Pi Zero** and **Pi 1**, you will need the [ARMv6](#) version of Node.js, so you will run the following commands:

```
wget https://nodejs.org/dist/v10.16.3/node-v10.16.3-linux-armv6l.tar.xz
tar xvJf node-v10.16.3-linux-armv6l.tar.xz
cd node-v10.16.3-linux-armv6l
sudo cp -R * /usr
sudo ln -s /usr/lib/node_modules /usr/lib/node
cd ..
rm -rf node-v10.16.3-linux-armv6l
```

For **Pi 2**, **Pi 3** and **Pi 4** models, the [ARMv7](#) version of Node.js is required, meaning that the bash commands are:

```
wget https://nodejs.org/dist/v14.15.1/node-v14.15.1-linux-armv7l.tar.xz
tar xvJf node-v14.15.1-linux-armv7l.tar.xz
cd node-v14.15.1-linux-armv7l
sudo cp -R * /usr
sudo ln -s /usr/lib/node_modules /usr/lib/node
cd ..
rm -rf node-v14.15.1-linux-armv7l
```


Install studio-supervisor

In order to install studio-supervisor, the following commands are required:

```
sudo su -
npm install -g --unsafe-perm studio-supervisor

exit
sudo mkdir /wyliodrin
```

Write the supervisor script

Using nano editor, write the /etc/supervisor/conf.d/studiosupervisor.conf file with the following contents:

To start the editor, type

```
sudo nano /etc/supervisor/conf.d/studio-supervisor.conf
```

```
[program:studio-supervisor]
command=/usr/bin/studio-supervisor raspberrypi
home=/wyliodrin
user=pi
```

Press Ctrl+X to save and exit the editor. Press Y when whether to save the file.

After that, you have to make the **/wyliodrin** directory your home directory:

```
sudo chown pi:pi /wyliodrin
cp /home/pi/.bashrc /wyliodrin/.bashrc
```

The final step is to refresh the board by running the command:

```
sudo supervisorctl reload
```

Install Docker manually

In order to install Docker, the following commands are required: .. code-block:: bash

```
sudo apt-get update && sudo apt-get upgrade curl -fsSL https://get.docker.com -o get-docker.shgit config
--global user.email "youremail@yourdomain" sudo sh get-docker.sh sudo usermod -aG docker pi
```

Now, you'll have to restart the board using: .. code-block:: bash

```
sudo reboot
```

To see if the installation worked, check the Docker version: .. code-block:: bash

```
docker version
```

Note: For **raspberry pi 0**, in order to work, after your first try to create a container, you have to go to the menu, select Use Advanced Mode and, in the dockerfile, change the default image with: FROM /balenalib/raspberry-pi-node:14.

2.1.4 Connecting via web

The connection of a Raspberry Pi board to the web version of Wyliodrin STUDIO demands an Internet connection and the creation of a file, **wyliodrin.json**, that will be written and stored on the SD card. The purpose of this configuration file is to keep a series of particular informations about the device and the platform, so the both instances be able to recognize and communicate with each other.

Acquiring the **wyliodrin.json** file assumes that you will have to launch the web version of the application and to click on the *Connect* button. After selecting the *New Device* option from the popup, a new dialog box will be opened and will ask you for the name of your new device.

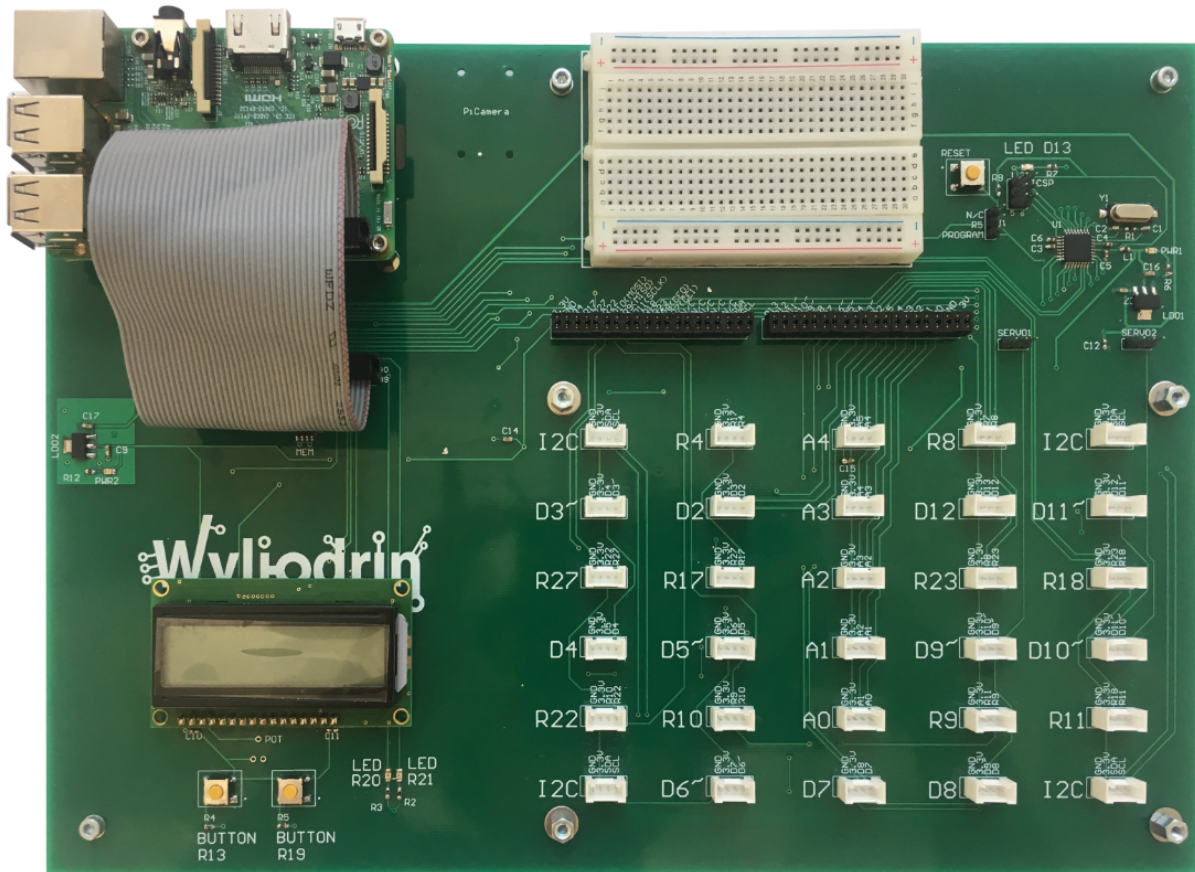
Once you start typing the name of your device, a JSON structure is automatically generated depending on the entered data. The format of the object consists of the following properties:

Property title	Description
<i>token</i>	unique identifier for the device, automatically assigned by the program
<i>id</i>	device name, updated as you change the name in the input box
<i>server</i>	endpoint

The content of this JSON structure has to be copied into a file that you will name **wyliodrin.json**, as mentioned before. Once the file created and saved, it has to be stored on the SD card, in the partition called **boot**. This action can be done by inserting the flashed card into your personal computer, which will lead to the automatic opening of the *boot* partition.

After copying the configuration file to the destination indicated, you can insert the SD card into the Raspberry Pi, connect the board to the Internet and power it on. At this step, if you hit the *Connect* button of the web application, you should see your Raspberry Pi device into the list of available devices and by clicking on its name you will be able to connect it to the IDE.

2.1.5 Wyliolab Board



If you are using the Wyliolab boards, you can download the pre-configured image for [Pi Zero](#) and [Pi 1](#), or the image for [Pi 2](#), [Pi 3](#) and [Pi 4](#).

If you choose to continue the manual setup for the Raspberry Pi of the Wyliolab board, you should run the following commands:

```
sudo pip3 install wyliozero

sudo su -
npm install -g --unsafe-perm studio-supervisor

exit
sudo nano /etc/supervisor/conf.d/studio-supervisor.conf
```

```
[program:studio-supervisor]
command=/usr/bin/studio-supervisor raspberrypi wyliolab
home=/wyliodrin
user=pi
```

After modifying the content of the *studio-supervisor.conf* file, you will have to run:

```
sudo raspi-config
```

In the prompt that will be opened, you will have to select the fifth option(Interfacing Options), then in the Configuration Tool section you will have to pick *P6 Serial* in order to disable the shell and enable the serial port.

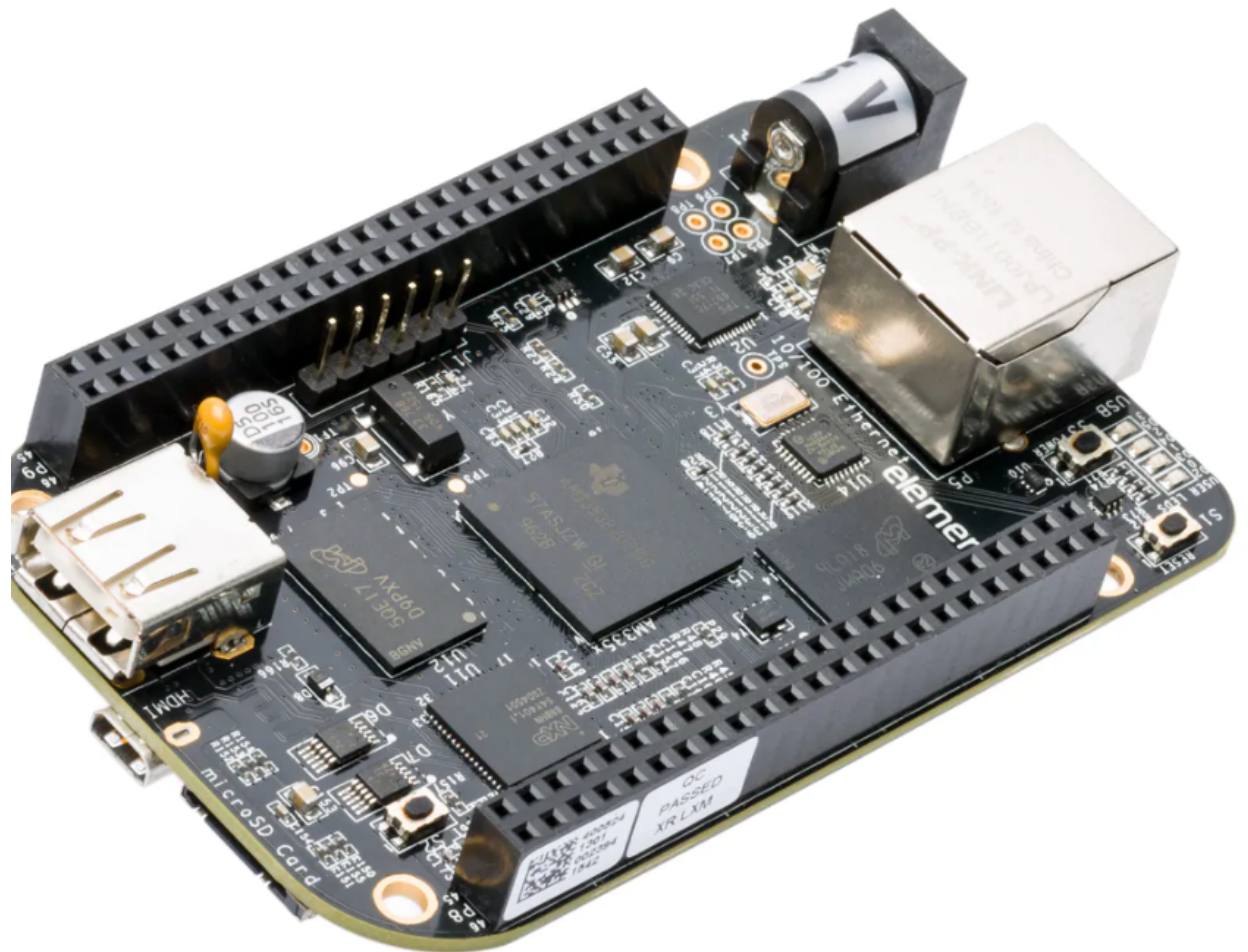
The final step before using the Wyliolab board is to reboot it.

2.1.6 Set up wireless

To set up your board wireless, please follow the steps in the link: [Set up wireless](#).

2.2 Beaglebone Black

This tutorial will show you how to set up a Beaglebone Black device.



2.2.1 Download the pre-configured image

The easiest way to set up a BeagleBone Black board so that it becomes available for Wyliodrin STUDIO is to download an image that is already configured.

Download the image for [BeagleBone Black](#).

Once the image downloaded and unzipped, the only thing that you have to do is to *flash* it. After that, you can simply insert the SD card into the BeagleBone Black and your board should be visible within Wyliodrin STUDIO.

2.2.2 Set up the board manually

However, you can also choose to configure the required image by yourself.

This will imply flashing an image with the OS (Debian), installing the STUDIO Supervisor container and setting up some configuration files.

Download the Debian image

You will need to:

1. Download a Debian Image
2. Install the Studio Supervisor
3. Setup a provisioning file

Download the [Debian IoT](#) image from the Beagle Board foundation. This is the standard OS for the BeagleBone Black provided by the manufacturer.

Flash the image

The downloaded image needs to be flash (written) to an SD card. The minimum size of the SD card is 4 GB.

Note: We recommend a minimum of 8 GB Class 10 SD Card. For small applications 4 GB might be enough.

To flash the image, you will need a special software. The recommended application is [Etcher](#).

Note: For Linux users, you may use the **dd** utility.

Install STUDIO Supervisor

To be able to access the Studio network, the BeagleBone Black needs to run the STUDIO Supervisor software. The following tutorial will explain how to install it.

After writing the SD Card, insert it into the board and start the device. You will have to access it. This can be done either by:

- connecting the BeagleBone Black to the network and use a SSH to connect to it
- connect a monitor and a keyboard to the board

If you are using SSH, you will have to input 192.168.7.2 as the host IP address and then login with the appropriate credentials:

username: *debian*

password: *temppwd*

Stop additional services

The BeagleBone Black image has several servers started. These are used mainly for development. Run the commands to stop them:

```
sudo systemctl disable bonescript.service
sudo systemctl disable bonescript-autorun.service
sudo systemctl disable bonescript.socket
sudo systemctl disable apache2
sudo systemctl disable cloud9.service
sudo systemctl disable cloud9.socket
sudo systemctl disable getty@tty1
sudo systemctl disable node-red.socket
```

Install Dependencies

The dependencies you will have to install are:

- *supervisor*: allows you to monitor processes related to a project
- *redis*: database management system
- *build-essential*: reference package for all the packages required for compilation
- *git*: required for the **npm install** command to download git included package
- *python3-pip*: python 3 programming language

```
sudo apt-get update
sudo apt-get install -y supervisor redis-server build-essential git python3-pip

# To enable the Notebook tab, you should also run
sudo pip3 install redis pygments
```

Install Node.js

The next step is to [install NodeJS](#).

For BeagleBone Black, the [ARMv7](#) version of Node.js is required, meaning that the bash commands are:

```
wget https://nodejs.org/dist/v10.16.3/node-v10.16.3-linux-armv7l.tar.xz
tar xvJf node-v10.16.3-linux-armv7l.tar.xz
```

After installing and unzipping Node, you should reboot the board and restart the session and remove old node:

```
sudo rm /usr/bin/npm
sudo rm /usr/bin/npm
sudo rm -f /usr/lib/node_modules
```

Continue the configuration by running the following commands:

```
cd node-v10.16.3-linux-armv7l
sudo cp -R * /usr
sudo ln -s /usr/lib/node_modules /usr/lib/node
cd ..
rm -rf node-v10.16.3-linux-armv7l
```

Install studio-supervisor

In order to install studio-supervisor, the following commands are required:

```
sudo su -
npm install -g --unsafe-perm studio-supervisor
exit
sudo mkdir /wyliodrin
```

Write the supervisor script

Using nano editor, write the `/etc/supervisor/conf.d/studiosupervisor.conf` file with the following contents:

To start the editor, type

```
sudo nano /etc/supervisor/conf.d/studio-supervisor.conf
```



```
[program:studio-supervisor]
command=/usr/bin/studio-supervisor beaglebone
home=/wyliodrin
user=debian
```

Press Ctrl+X to save and exit the editor. Press Y when whether to save the file.

After that, you have to make the **/wyliodrin** directory your home directory:

```
sudo chown debian:debian /wyliodrin
cp /home/debian/.bashrc /wyliodrin/.bashrc
```

Note: While using the Pico-Pi device, you will need to run some commands as root, meaning that each time you will use **sudo**, the system will ask you to input the password. In order to be able to run the **sudo** command without entering a password, you will have to configure a setting.

You will have to run the **sudo visudo** command, which will open the *etc/sudoers* file. You will have to modify the content by moving the next line at the end of the file:

```
debian ALL=(ALL) NOPASSWD: ALL
```

The final step is to refresh the board by running the command:

```
sudo supervisorctl reload
```

2.2.3 Connecting via web

The connection of a BeagleBone Black board to the web version of Wyliodrin STUDIO demands an Internet connection and the creation of a file, **wyliodrin.json**, that will be written and stored on the SD card. The purpose of this configuration file is to keep a series of particular informations about the device and the platform, so the both instances be able to recognize and communicate with each other.

Acquiring the **wyliodrin.json** file assumes that you will have to launch the web version of the application and to click on the *Connect* button. After selecting the *New Device* option from the popup, a new dialog box will be opened and will ask you for the name of your new device.

Once you start typing the name of your device, a JSON structure is automatically generated depending on the entered data. The format of the object consists of the following properties:

Property title	Description
<i>token</i>	unique identifier for the device, automatically assigned by the program
<i>id</i>	device name, updated as you change the name in the input box
<i>server</i>	endpoint

The content of this JSON structure has to be copied into a file that you will name **wyliodrin.json**, as mentioned before.

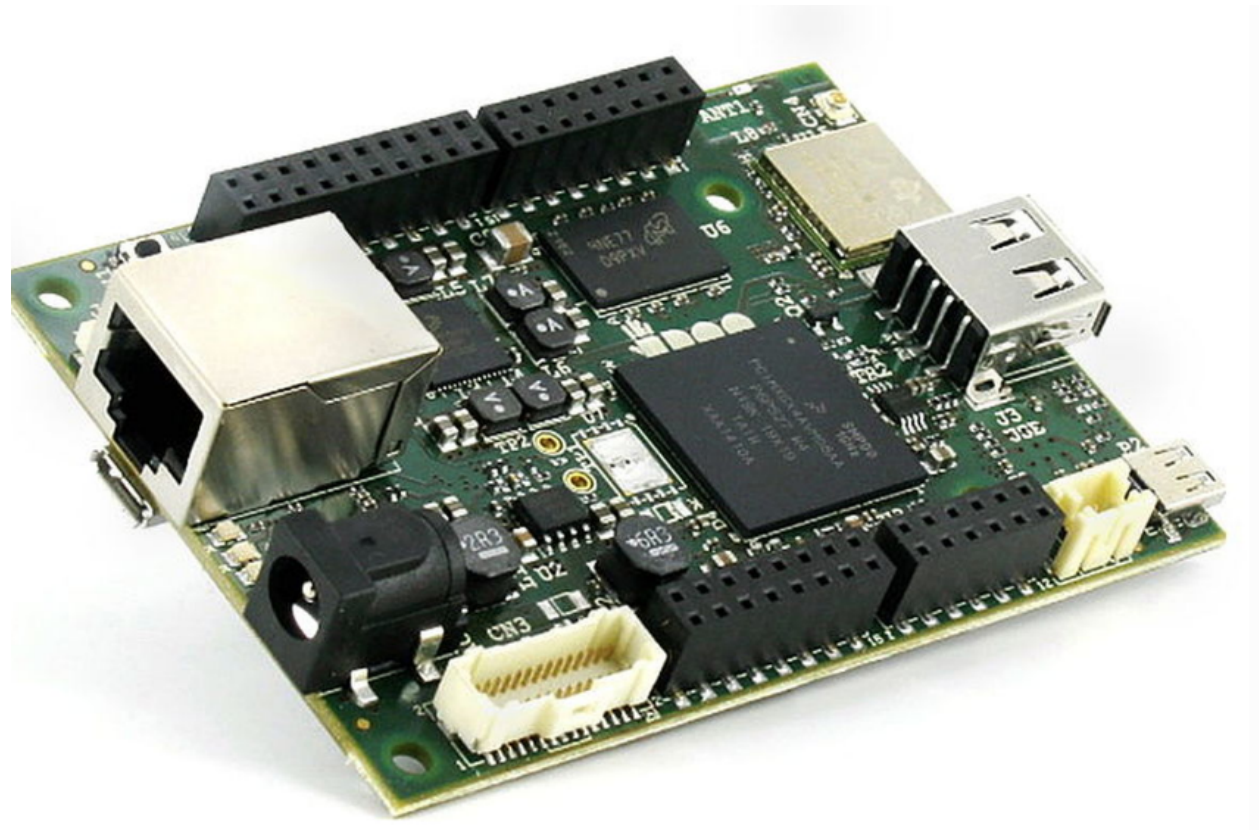
To add this file, you will have to connect the device to Wylodrin STUDIO, open the **Shell** tab and run:

```
sudo nano /boot/wylodrin.json
```

After creating the configuration file to the destination indicated, you can hit the *Connect* button of the web application. At this point, you should see your BeagleBone Black device into the list of available devices and by clicking on its name you will be able to connect it to the IDE.

2.3 Udo Neo

This tutorial will show you how to set up a Udo Neo device.



2.3.1 Set up the board manually

You can choose to configure the required image by yourself.

This will imply flashing an image with the OS (Ubuntu), installing the STUDIO Supervisor container and setting up some configuration files.

Download the Ubuntu image

You will need to:

1. Download a Ubuntu Image
2. Install the Studio Supervisor
3. Setup a provisioning file

Download the [Ubuntu 16](#) image for Udoo Neo.

Flash the image

The downloaded image needs to be flash (written) to an SD card. The minimum size of the SD card is 4 GB.

Note: We recommend a minimum of 8 GB Class 10 SD Card. For small applications 4 GB might be enough.

To flash the image, you will need a special software. The recommended application is [Etcher](#).

Note: For Linux users, you may use the **dd** utility.

Install STUDIO Supervisor

To be able to access the Studio network, the Udoo Neo needs to run the STUDIO Supervisor software. The following tutorial will explain how to install it.

After writing the SD Card, insert it into the board and start the device. You will have to access it. This can be done either by:

- connecting the Udoo Neo to the network and use a SSH to connect to it
- connect a monitor and a keyboard to the board

If you are using SSH, you will have to input 192.168.7.2 as the host IP address and then login with the appropriate credentials:

username: *udooer*

password: *udooer*

Install Dependencies

The dependencies you will have to install are:

- *supervisor*: allows you to monitor processes related to a project
- *redis*: database management system

- *build-essential*: reference package for all the packages required for compilation
- *git*: required for the **npm install** command to download git included package
- *python3-pip*: python 3 programming language

```
sudo apt-get update
sudo apt-get install -y supervisor redis-server build-essential git python3-pip

# To enable the Notebook tab, you should also run
sudo pip3 install redis pygments
```

Install Node.js

The next step is to install NodeJS.

For Udoo Neo, the [ARMv7](#) version of Node.js is required, meaning that the bash commands are:

```
wget https://nodejs.org/dist/v10.16.3/node-v10.16.3-linux-armv7l.tar.xz
tar xvJf node-v10.16.3-linux-armv7l.tar.xz
```

After installing and unzipping Node, you should reboot the board and restart the session and remove old node:

```
sudo rm /usr/bin/npm
sudo rm /usr/bin/node
sudo rm /usr/lib/node_modules
```

Continue the configuration by running the following commands:

```
cd node-v10.16.3-linux-armv7l
sudo cp -R * /usr
sudo ln -s /usr/lib/node_modules /usr/lib/node
cd ..
rm -rf node-v10.16.3-linux-armv7l
```

Install studio-supervisor

In order to install studio-supervisor, the following commands are required:

```
sudo su -
npm install -g --unsafe-perm studio-supervisor
```

(continues on next page)

(continued from previous page)

```
exit
sudo mkdir /wyliodrin
```

Write the supervisor script

Using nano editor, write the `/etc/supervisor/conf.d/studiosupervisor.conf` file with the following contents:

To start the editor, type

```
sudo nano /etc/supervisor/conf.d/studio-supervisor.conf
```

```
[program:studio-supervisor]
command=/usr/bin/studio-supervisor udooneo
home=/wyliodrin
user=udooer
```

Press Ctrl+X to save and exit the editor. Press Y when whether to save the file.

After that, you have to make the `/wyliodrin` directory your home directory:

```
sudo chown udooer:udooer /wyliodrin
cp /home/udooer/.bashrc /wyliodrin/.bashrc
```

The final step is to refresh the board by running the command:

```
sudo supervisorctl reload
```

2.3.2 Connecting via web

The connection of a Udo Neo board to the web version of Wyliodrin STUDIO demands an Internet connection and the creation of a file, **wyliodrin.json**, that will be written and stored on the SD card. The purpose of this configuration file is to keep a series of particular informations about the device and the platform, so the both instances be able to recognize and communicate with each other.

Acquiring the **wyliodrin.json** file assumes that you will have to launch the web version of the application and to click on the *Connect* button. After selecting the *New Device* option from the popup, a new dialog box will be opened and will ask you for the name of your new device.

Once you start typing the name of your device, a JSON structure is automatically generated depending on the entered data. The format of the object consists of the following properties:

Property title	Description
<i>token</i>	unique identifier for the device, automatically assigned by the program
<i>id</i>	device name, updated as you change the name in the input box
<i>server</i>	endpoint

The content of this JSON structure has to be copied into a file that you will name **wylidrin.json**, as mentioned before.

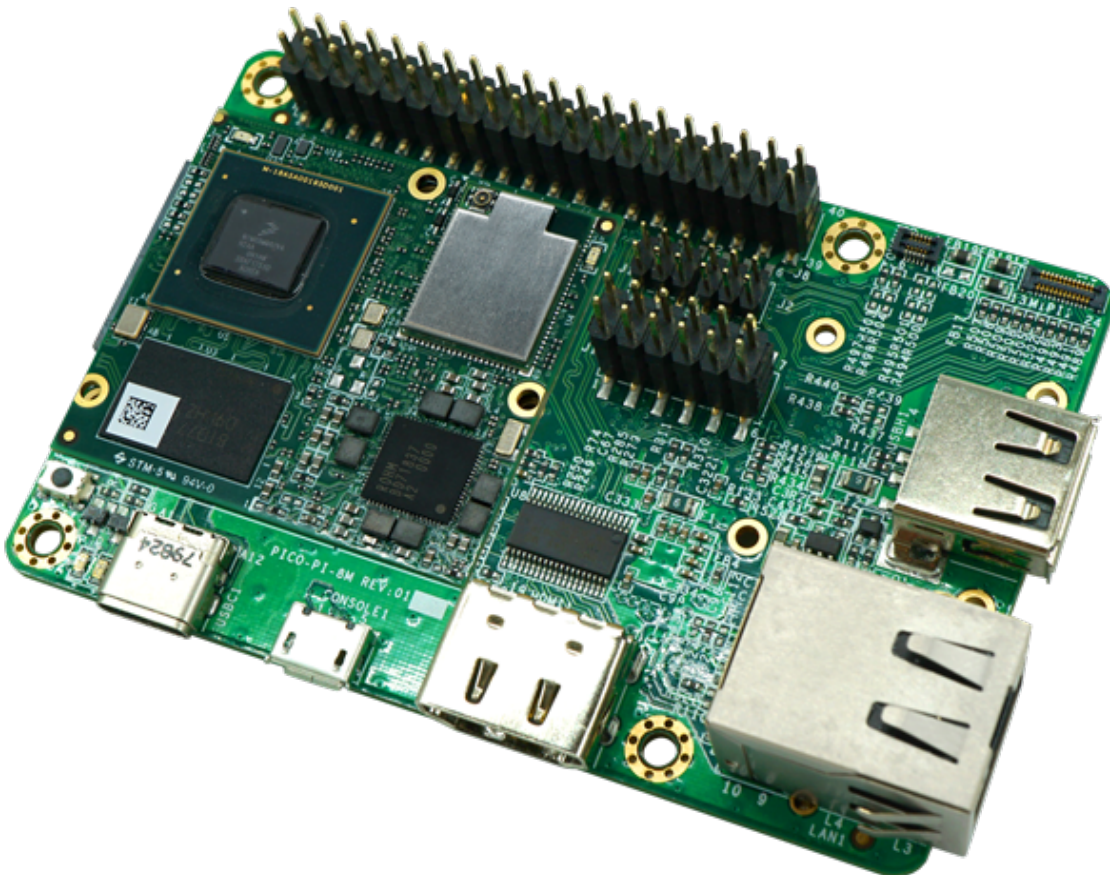
To add this file, you will have to connect the device to Wylidrin STUDIO, open the **Shell** tab and run:

```
sudo nano /boot/wylidrin.json
```

After creating the configuration file to the destination indicated, you can hit the *Connect* button of the web application. At this point, you should see your Udoo Neo device into the list of available devices and by clicking on its name you will be able to connect it to the IDE.

2.4 Pico-Pi

This will show how to set up a Pico-Pi device.



To configure the Pico-Pi IMX8M board, it will be necessary to flash an image with the Ubuntu operating system, install the Studio-Supervisor container and set up some configuration files.

2.4.1 Download the pre-configured image

The easiest way to set up a Pico-Pi IMX8M board so that it becomes available for Wyliodrin STUDIO is to download an image that is already configured.

Download the image for [PicoPi IMX8M](#).

Once the image downloaded and unzipped, the only thing that you have to do is to *flash* it. After that, your Pico-Pi board should be visible within Wyliodrin STUDIO.

2.4.2 Set up the board manually

Enable the USB mass storage device

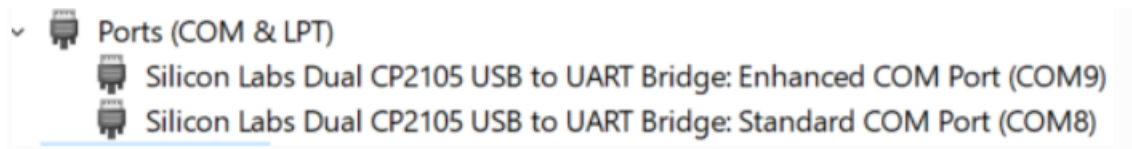
The first step is to connect the Pico-Pi device directly to your computer, using the micro USB and USB type C cables.

If your computer is running on **Linux**, you should be able to see the

If you are using **Windows**, you will need an additional driver to see the COM ports:

<https://www.silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers>

After downloading and extracting the files, you should open the Device Manager, right click on the Ports section and select the driver. By the end, you should be able to see the following devices:



Export the EMMC device as mass storage to the host computer

1. Set up the serial terminal

As the Pico-Pi is already directly connected to your computer, you have to get a serial terminal program running. For Linux, we suggest you to use **screen**, but any other serial terminal should work.

If you are using Windows, we recommend you to download and open [Putty](#) and customize the session with the following options:

Connection type	Serial
Serial line	COM port for Pico-Pi, in this example COM9
Speed	115200

Once the session started, it will load U-boot and you will be able to see the text “Hit any key to stop autoboot:”. Pressing on a key will stop the boot process and open a boot prompt.

Note: If the boot prompt doesn't appear, you should reboot the board by pressing the Restart button.

2. List the accessible devices

In order to get a list with the MMC devices, you should run the following command:

```
mmc list
```

The output should look like this:

3. Export the EMMC device

To export the Pico-Pi device to the host computer, you will run the next command:

```
ums 0 mmc 0
```

The output will be:

```
UMS: LUN 0, dev 1, hwpart 0, sector 0x0, count 0xe90000  
/
```

A rotating cursor will be visible while the USB Mass Storage is running and the boot prompt can be exited by pressing CTRL+C.

If you followed this steps, a new USB device should appear on your PC and you will use it to load the Ubuntu image.

Load the image into EMMC

Download the [Ubuntu](#) image from the TechNexion foundation. This is the standard OS for the Pico-Pi IMX8M provided by the manufacturer.

Flash the Ubuntu image

The downloaded image needs to be flash (written) directly to the Pico Pi.

To flash the image, you will need a special software. The recommended application is [Etcher](#).

Once the Ubuntu image flashed on your Pico-Pi board, you will have to reboot the device by pressing on its Restart button and wait for it to boot the Ubuntu OS without pressing any key. When the boot process is finished, you will be asked to provide the login credentials. For this type of device, the login username is *ubuntu*, same as the password, *ubuntu*.

Install STUDIO Supervisor

To be able to access the Studio network, the Pico-Pi needs to run the STUDIO Supervisor software. The following tutorial will explain how to install it.

After writing the image on the device, you will have to connect the Pico-Pi to the network and use a SSH to connect to it.

Install Dependencies

The dependencies you will have to install are:

- *supervisor*: allows you to monitor processes related to a project
- *redis*: database management system
- *build-essential*: reference package for all the packages required for compilation
- *git*: required for the **npm install** command to download git included package
- *python3-pip*: python 3 programming language

```
sudo apt-get update
sudo apt-get install -y supervisor redis build-essential git python3-pip

# To enable the Notebook tab, you should also run
sudo pip3 install redis pygments
```

Install Node.js

The next step is to [install NodeJS](#).

For the Pico-Pi IMX8M you will need the [ARMv8](#) version of Node.js, so you will run the following commands:

```
sudo apt-get install wget
wget https://nodejs.org/dist/v10.16.3/node-v10.16.3-linux-arm64.tar.xz

tar xvJf node-v10.16.3-linux-arm64.tar.xz

cd node-v10.16.3-linux-arm64

sudo cp -R * /usr

sudo ln -s /usr/lib/node_modules /usr/lib/node

cd ..

rm -rf node-v10.16.3-linux-arm64
```


Install studio-supervisor

In order to install studio-supervisor, the following commands are required:

```
sudo su -
npm install -g --unsafe-perm studio-supervisor

exit
sudo mkdir /wyliodrin
```

Write the supervisor script

Using nano editor, write the `/etc/supervisor/conf.d/studiosupervisor.conf` file with the following contents:

To start the editor, type

```
sudo apt-get install nano
sudo nano /etc/supervisor/conf.d/studio-supervisor.conf
```

```
[program:studio-supervisor]
command=/usr/bin/studio-supervisor picopi
home=/wyliodrin
user=ubuntu
```

Press Ctrl+X to save and exit the editor. Press Y when whether to save the file.

After that, you have to make the **/wyliodrin** directory your home directory:

```
sudo chown ubuntu:ubuntu /wyliodrin
cp /home/ubuntu/.bashrc /wyliodrin/.bashrc
```

Note: While using the Pico-Pi device, you will need to run some commands as root, meaning that each time you will use **sudo**, the system will ask you to input the password. In order to be able to run the sudo command without entering a password, you will have to configure a setting.

You will have to run the **sudo visudo** command, which will open the `etc/sudoers` file. You will have to modify the content by moving the next line at the end of the file:

```
ubuntu ALL=(ALL) NOPASSWD: ALL
```

If you are using Wyliodrin STUDIO locally, you will need to install the following utilities:

```
sudo apt-get install avahi-daemon
sudo apt-get install openssh-server
```

The final step is to refresh the board by running the command:

```
sudo supervisorctl reload
```

2.4.3 Connecting via web

The connection of a Pico-Pi IMX8M board to the web version of Wyliodrin STUDIO demands an Internet connection and the creation of a file, **wyliodrin.json**, that will be written and stored on the device. The purpose of this configuration file is to keep a series of particular informations about the device and the platform, so the both instances be able to recognize and communicate with each other.

Acquiring the **wyliodrin.json** file assumes that you will have to launch the web version of the application and to click on the *Connect* button. After selecting the *New Device* option from the popup, a new dialog box will be opened and will ask you for the name of your new device.

Once you start typing the name of your device, a JSON structure is automatically generated depending on the entered data. The format of the object consists of the following properties:

Property title	Description
<i>token</i>	unique identifier for the device, automatically assigned by the program
<i>id</i>	device name, updated as you change the name in the input box
<i>server</i>	endpoint

The content of this JSON structure has to be copied into a file that you will name **wyliodrin.json**, as mentioned before. Once the file created and saved, it has to be stored on **boot** partition of your Pico-Pi.

To mount the boot partition, you will have to run the following command:

```
sudo nano /etc/fstab
```

You will have to add the following text content within the *fstab* file:

```
/dev/mmcblk0p1 /boot auto ro 0 0
```

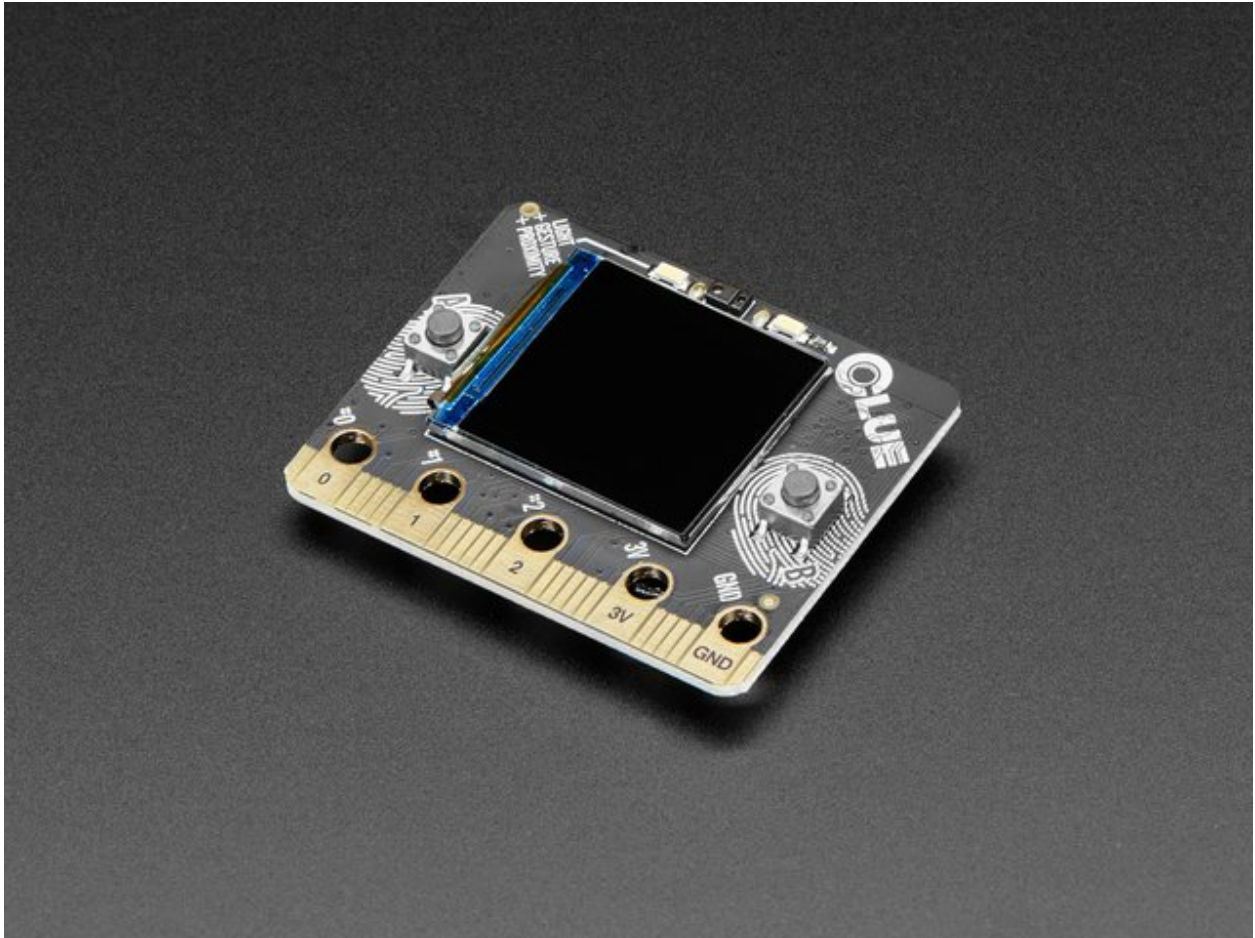
After copying the configuration file to the destination indicated, you can reboot your board using the Restart button. At this step, if you hit the *Connect* button of the web application, you should see your Pico-Pi device into the list of available devices and by clicking on its name you will be able to connect it to the IDE.

2.5 Adafruit CLUE (CircuitPython)

Adafruit CLUE is a board similar to the [Micro:Bit](#), but based on the Nordic nRF52840 SoC. It includes the following sensors:

- 9 axis inertial LSM6DS33 + LIS3MDL sensor
- humidity and temperature sensor
- barometric sensor
- microphone
- gesture, proximity, light color and light intensity sensor

It also has a 240x240 LCD display.



This documentation describes how to use the [Adafruit CLUE](#) with [CircuitPython](#).

Adafruit CLUE can be used with both the offline and the web version (Google Chrome only) of WyliodrinSTUDIO.

2.5.1 Installing CircuitPython

You will have to follow these steps:

1. Download CircuitPython
2. Flash CircuitPython to the board
3. Load the libraries

Download CircuitPython

CircuitPython is an Adafruit modified version of MicroPython. Adafruit provides a downloadable image for several boards. You have to download the [Adafruit CLUE CircuitPython image](#).

Please download the latest stable version. This should be a UF2 file.

Flash the CircuitPython

The UF2 file that you have downloaded at the previous step has to be written to the board. Adafruit has provided a very easy method to do that. Connect your board to your computer using a USB cable and double press the button on the back of the board (the part that does not have a display). The Neopixel LED should start flashing green.

Double pressing the button on the back will put the board into DFU mode. This will display connect to your computer a USB drive called BOOT. Copy and paste the downloaded UF2 file to this drive. This will flash CircuitPython to the board.

2.5.2 Offline WyliodrinSTUDIO

Connect the board to your computer using the USB cable. Run Wyliodrin STUDIO and open the Connect menu. You should have an option called Adafruit Industries or Adafruit CLUE. Select that board. A popup with some options will appear, just use the default options and click Connect.

You should be connected to the board.

2.5.3 Web WyliodrinSTUDIO

Note: Using the web version requires Google Chrome with some experimental features enabled

To use the Adafruit CLUE in the web version, you will have to use Google Chrome and enable Experimental Features.

To enable Experimental Features in Google Chrome, follow the steps:

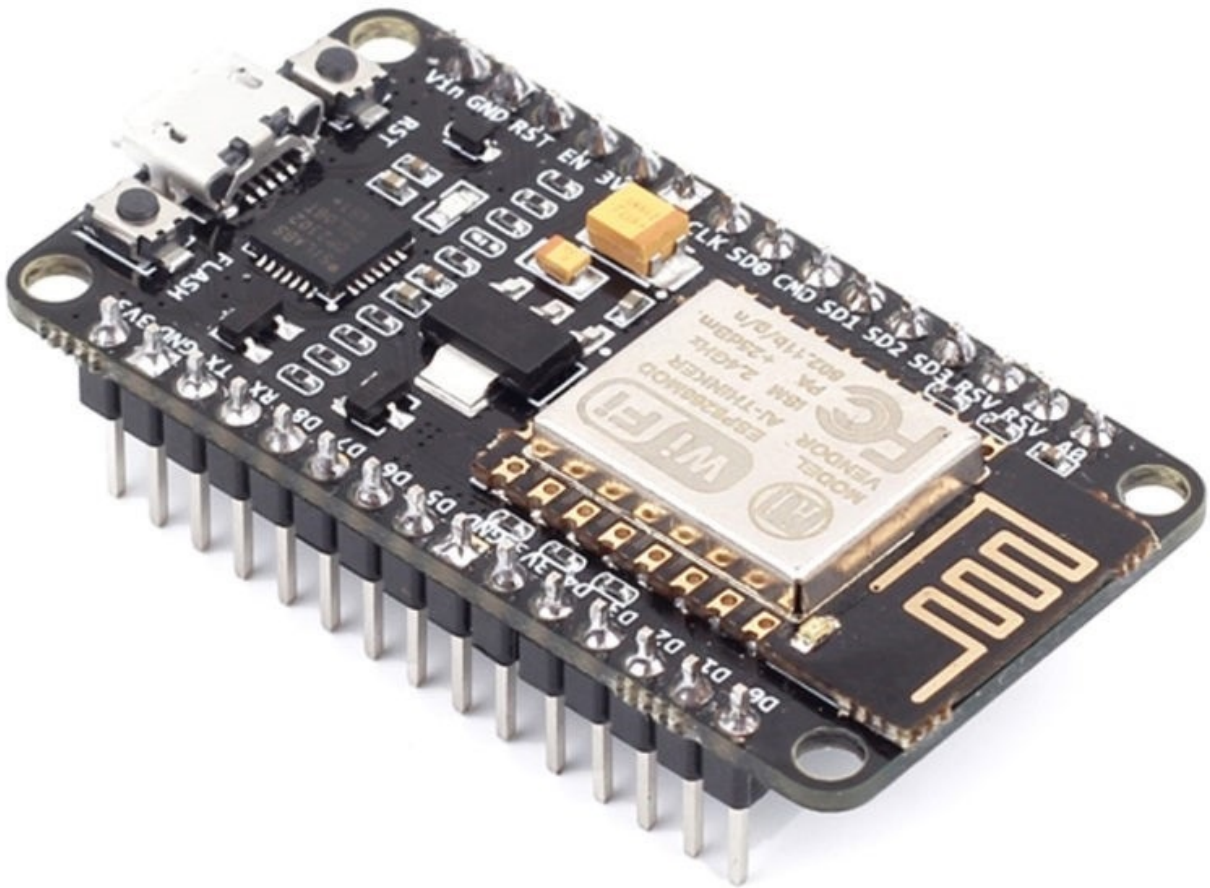
1. In the Chrome search bar write *chrome://flags*
2. Search the search bar for the flags: *#enable-experimental-web-platform-features*
3. Set the ENABLE flag for *Experimental Web Platform features*
4. At the bottom right click RELAUNCH button
5. Restart the browser

After enabling Experimental Features, connect the board to your computer using the USB cable and click the Connect menu. Select the MicroPython option. A popup will appear, you can safely use the default settings and click Connect. The browser will ask you to select the serial port. Select the port that has the *Adafruit* word in its name.

You should be connected to the board.

2.6 ESP 8266 (MicroPython)

This will show how to set up a ESP8266 device.



2.6.1 Windows

1. Download Micropython firmware

Open a browser and type this link: <https://micropython.org/download/esp8266/>, then go install the latest version (without opening it).

Suggestion: create a folder on your computer named “esp8266” or “micropython” and download it there.

Stable firmware, 2M or more of flash

The following files are stable firmware for the ESP8266. Program your board using the `esptool.py` program as described [in the tutorial](#).

Note: v1.13 of the firmware has a new flash filesystem layout, and uses `littlefs` as the filesystem by default. When upgrading from older firmware please backup your files first, and either erase all flash before upgrading, or after upgrading execute `uos.VfsLfs2.mkfs(bdev)`. Also note that v1.12 and earlier will work on modules with 1M or more of flash, while v1.13 requires 2M or more.

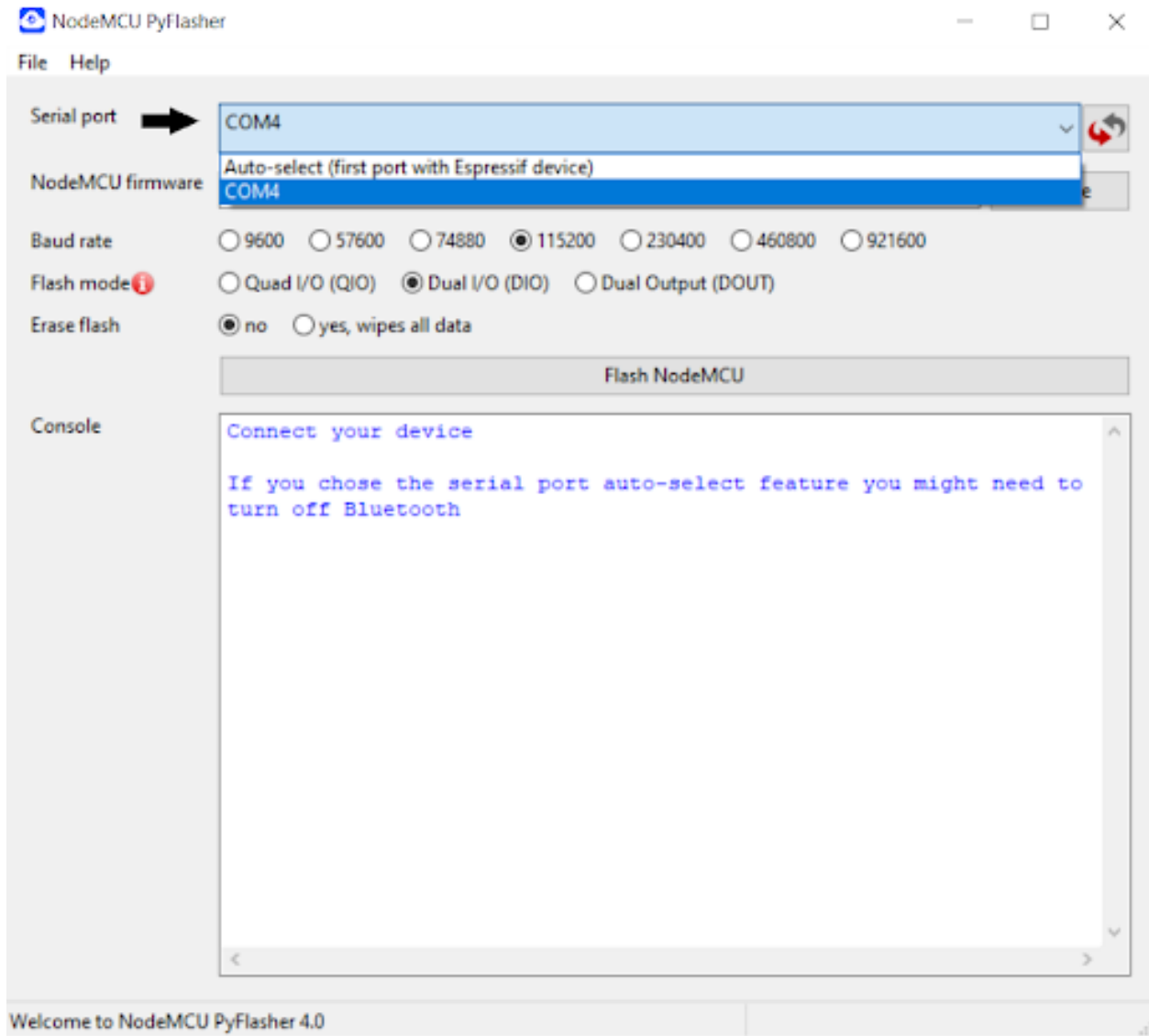
- [esp8266-20200911-v1.13.bin](#) (elf, map) (latest)

2. Install NodeMCU PyFlasher and flashing Micropython on ESP8266

NodeMCU PyFlasher is a new GUI tool to flash NodeMCU based on `esptool.py` and `wxPython`. It is available for Windows and for macOS. First, you have to connect the ESP8266 to your computer. Take the USB cable from the kit and Put the USB-C in one of your ports and then connect the micro-USB to the microcontroller.

Check this link [install NodeMCU](#), scroll down to the executables. If you have Windows 10 choose the first one (*NodeMCU-PyFlasher-4.0-x64.exe*), if you have Windows 7 choose the second one (*NodeMCU-PyFlasher-4.0-x86.exe*).

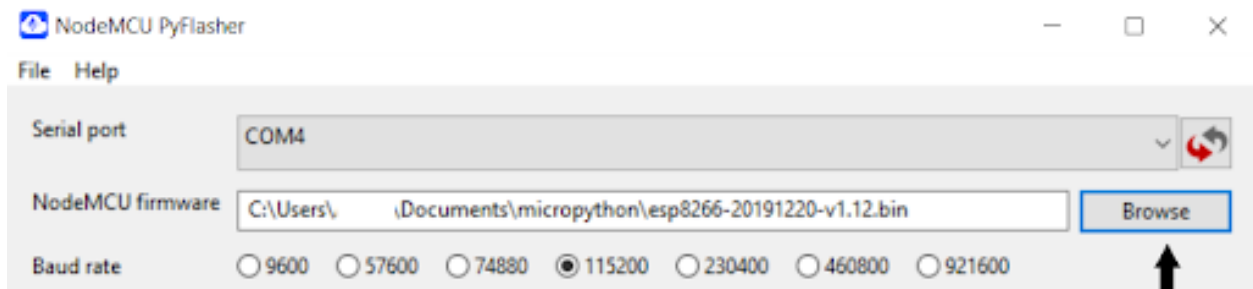
After the installation will be completed, click on the NodeMCU PyFlasher .exe file and you should have a similar window:



First, you have to choose your serial port, in this case it is “COM4”, (the number after “COM” is based on the port that you chose) like in the image below.

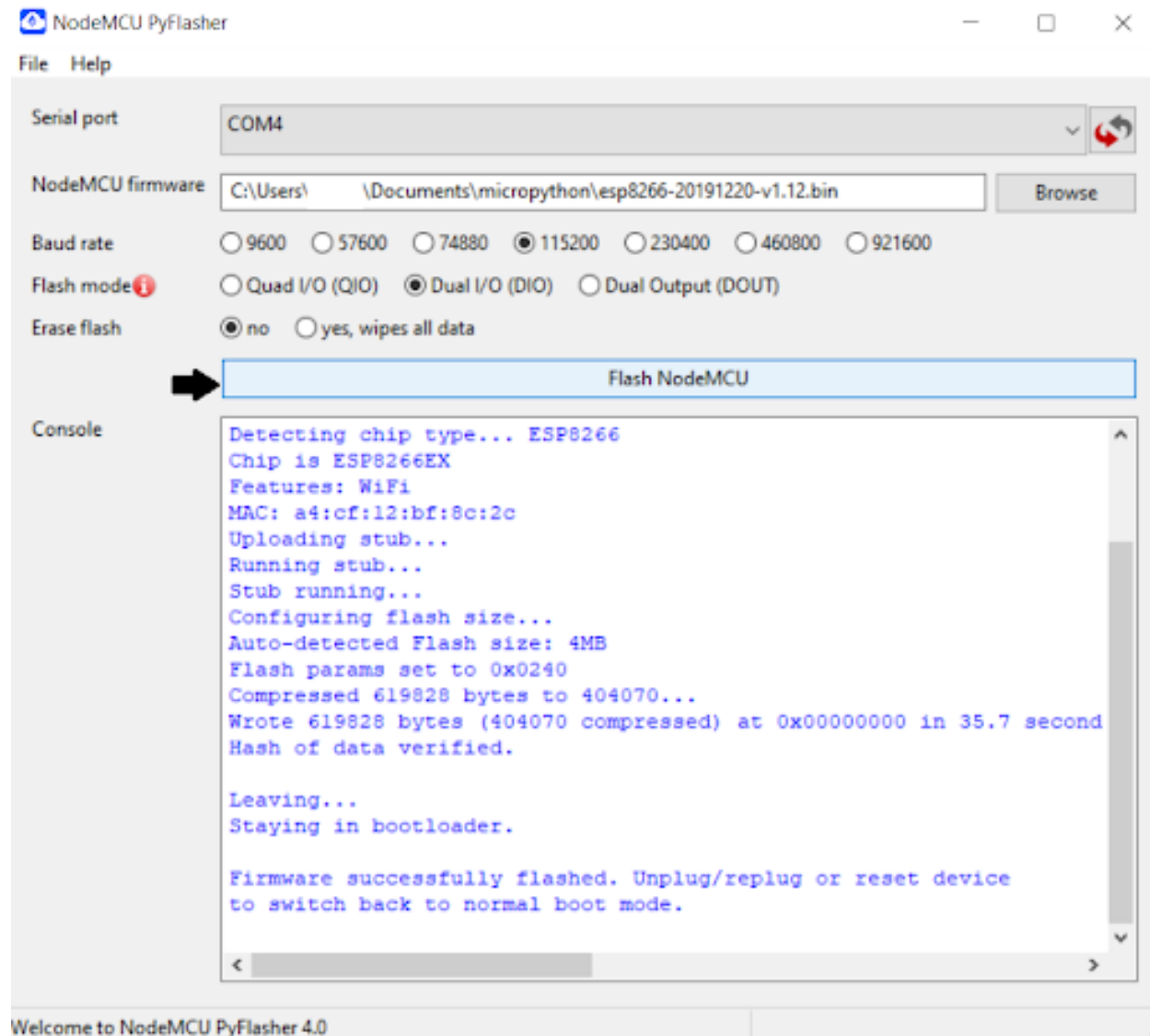
If you want to check which port you chose go in the Open Start menu and type “Device Manager”, then check Ports section (the number after “COM” is based on the port that you chose).

Succeeding, you have to click on “Browse” and go to the folder where you installed Micropython and choose the .bin file. Next you should chose the “Baud rate” like in the image below:



If you encounter errors, you need to reduce the baud rate (for example 9600 or up down). “1115200” is the speed read by serial port

As final step, you have to click on the button “Flash NodeMCU”.



Congratulations, now you have Micropython on your ESP8266!

2.6.2 Linux

2.1 Install Python

First, you should check if you have python3 installed. For that open Terminal and type:

```
$ python3 --version
```


If the python version appears, you can skip the installation go to Verify PIP is installed. You might have a newer version of python

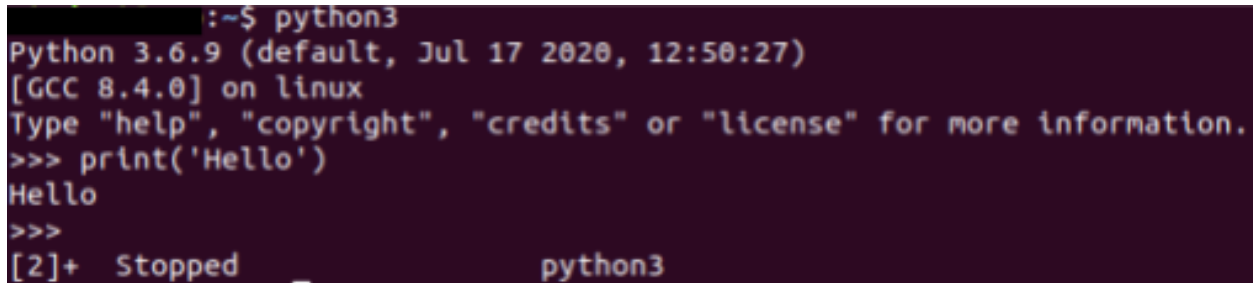
If you do not have python installed, you have to use this command:

```
$ sudo apt install python3
```

At the moment, you should have python3. In order to check if the installation is completed, type:

```
$ python3
```

Then try to code in python, like in the image below:



```

:~$ python3
Python 3.6.9 (default, Jul 17 2020, 12:50:27)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello')
Hello
>>>
[2]+  Stopped                  python3

```

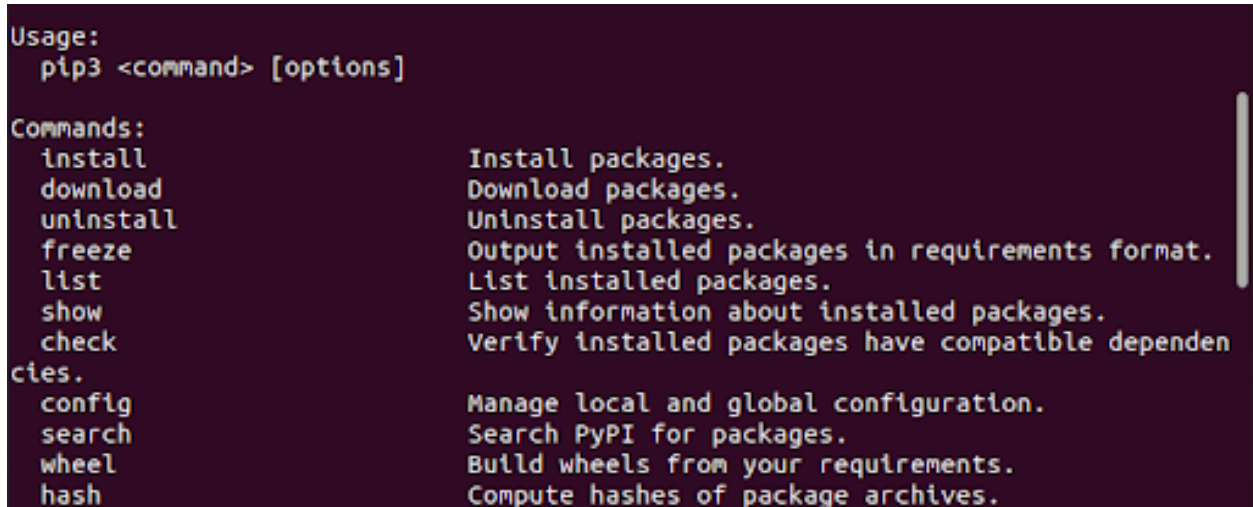
To exit python press *Ctrl+Z*.

Verify PIP is installed

Open Terminal and type :

```
$ pip3
```

If it is installed you should have a similar output :



```

Usage:
  pip3 <command> [options]

Commands:
  install           Install packages.
  download          Download packages.
  uninstall         Uninstall packages.
  freeze            Output installed packages in requirements format.
  list              List installed packages.
  show              Show information about installed packages.
  check             Verify installed packages have compatible dependen
cies.
  config            Manage local and global configuration.
  search            Search PyPI for packages.
  wheel             Build wheels from your requirements.
  hash              Compute hashes of package archives.

```

But if it has not been installed, you have to use the commands:

```
$ sudo apt-get update
```

```
$ sudo apt-get install python-pip
```

```
$ sudo pip install --upgrade pip
```

Congratulations, now you have installed Python!

2.2 Download Micropython firmware

Open a browser and type this link: <https://micropython.org/download/esp8266/>, then go install the latest version (without opening it).

Suggestion: create a folder on your computer named “esp8266” or “micropython” and download it there.

Stable firmware, 2M or more of flash

The following files are stable firmware for the ESP8266. Program your board using the `esptool.py` program as described in the tutorial.

Note: v1.13 of the firmware has a new flash filesystem layout, and uses `littlefs` as the filesystem by default. When upgrading from older firmware please backup your files first, and either erase all flash before upgrading, or after upgrading execute `uos.VfsLfs2.mkfs(bdev)`. Also note that v1.12 and earlier will work on modules with 1M or more of flash, while v1.13 requires 2M or more.

- [esp8266-20200911-v1.13.bin](#) (elf, map) (latest)

3. Flashing Micropython on ESP8266

Open Terminal and use the command:

```
$ pip3 install esptool
```

Then check the esptool installation by typing:

```
$ esptool
```

Connect the ESP8266 to your computer. Take the USB cable from the kit and Put the USB-C in one of your ports and then connect the micro-USB to the microcontroller.

Succeeding, go to the folder where you installed Micropython firmware. Use `$ls` command to list files and directories and `$ cd` to change the current working directory.

Type `$ dmesg` to see the port, you should have a similar output:

```

[30487.906511] raid6: .... xor() 20819 MB/s, rmw enabled
[30487.906512] raid6: using avx2x2 recovery algorithm
[30487.931305] xor: automatically using best checksumming function   avx
[30487.969520] Btrfs loaded, crc32c=crc32c-intel
[30766.632232] usb 2-2.1: USB disconnect, device number 22
[30766.872764] usb 2-2.1: new full-speed USB device number 23 using uhci_hcd
[30767.017016] usb 2-2.1: New USB device found, idVendor=0e0f, idProduct=0008, b
cdDevice= 1.00
[30767.017017] usb 2-2.1: New USB device strings: Mfr=1, Product=2, SerialNumber
=3
[30767.017019] usb 2-2.1: Product: Virtual Bluetooth Adapter
[30767.017019] usb 2-2.1: Manufacturer: VMware
[30767.017020] usb 2-2.1: SerialNumber: 000650268328
[30783.090649] usb 2-2.2: new full-speed USB device number 24 using uhci_hcd
[30783.408598] usb 2-2.2: New USB device found, idVendor=1a86, idProduct=7523, b
cdDevice= 2.54
[30783.408605] usb 2-2.2: New USB device strings: Mfr=0, Product=2, SerialNumber
=0
[30783.408609] usb 2-2.2: Product: USB2.0-Serial
[30783.473407] usbcore: registered new interface driver ch341
[30783.474111] usbserial: USB Serial support registered for ch341-uart
[30783.474464] ch341 2-2.2:1.0: ch341-uart converter detected
[30783.494868] usb 2-2.2: ch341-uart converter now attached to ttyUSB0

```

In this case the port is `ttyUSB0`.

After, use the command:

```
$ esptool.py --port /dev/ttyUSB0 erase_flash
```

for erasing the flash memory on the board. Instead of `ttyUSB0` you might have another port. You have to put the one that you have seen earlier.

Press the reset (RST) button from your ESP8266, then use the command:

```
$ esptool.py --port /dev/ttyUSB0 --baud 460800 write_flash --flash_size=detect 0 esp8266-20170108-
v1.8.7.bin
```

Keep in mind to put the port that you used in the previous command and pay attention to the version of MicroPython that you have installed. Instead of “`esp8266-20170108-v1.8.7.bin`” you might have another version. You must replace it in the command. If you encounter errors, you need to reduce the baud rate (for example 115200 or up down).

Next, connect to the serial console with command:

```
$ screen /dev/ttyUSB0 115200
```

“115200” is the speed read by serial port. To close it type **Ctrl+D** or **Ctrl+a** followed by **Ctrl+**.

Congratulations, now you have MicroPython on your ESP8266!

2.6.3 macOS

1. Download Micropython firmware

Open a browser and type this link: <https://micropython.org/download/esp8266/>, then go install the latest version (without opening it).

Suggestion: create a folder on your computer named “esp8266” or “micropython” and download it there.

Stable firmware, 2M or more of flash

The following files are stable firmware for the ESP8266. Program your board using the esptool.py program as described [in the tutorial](#).

Note: v1.13 of the firmware has a new flash filesystem layout, and uses littlefs as the filesystem by default. When upgrading from older firmware please backup your files first, and either erase all flash before upgrading, or after upgrading execute `uos.VfsLfs2.mkfs(bdev)`. Also note that v1.12 and earlier will work on modules with 1M or more of flash, while v1.13 requires 2M or more.

- [esp8266-20200911-v1.13.bin](#) (elf, map) (latest)

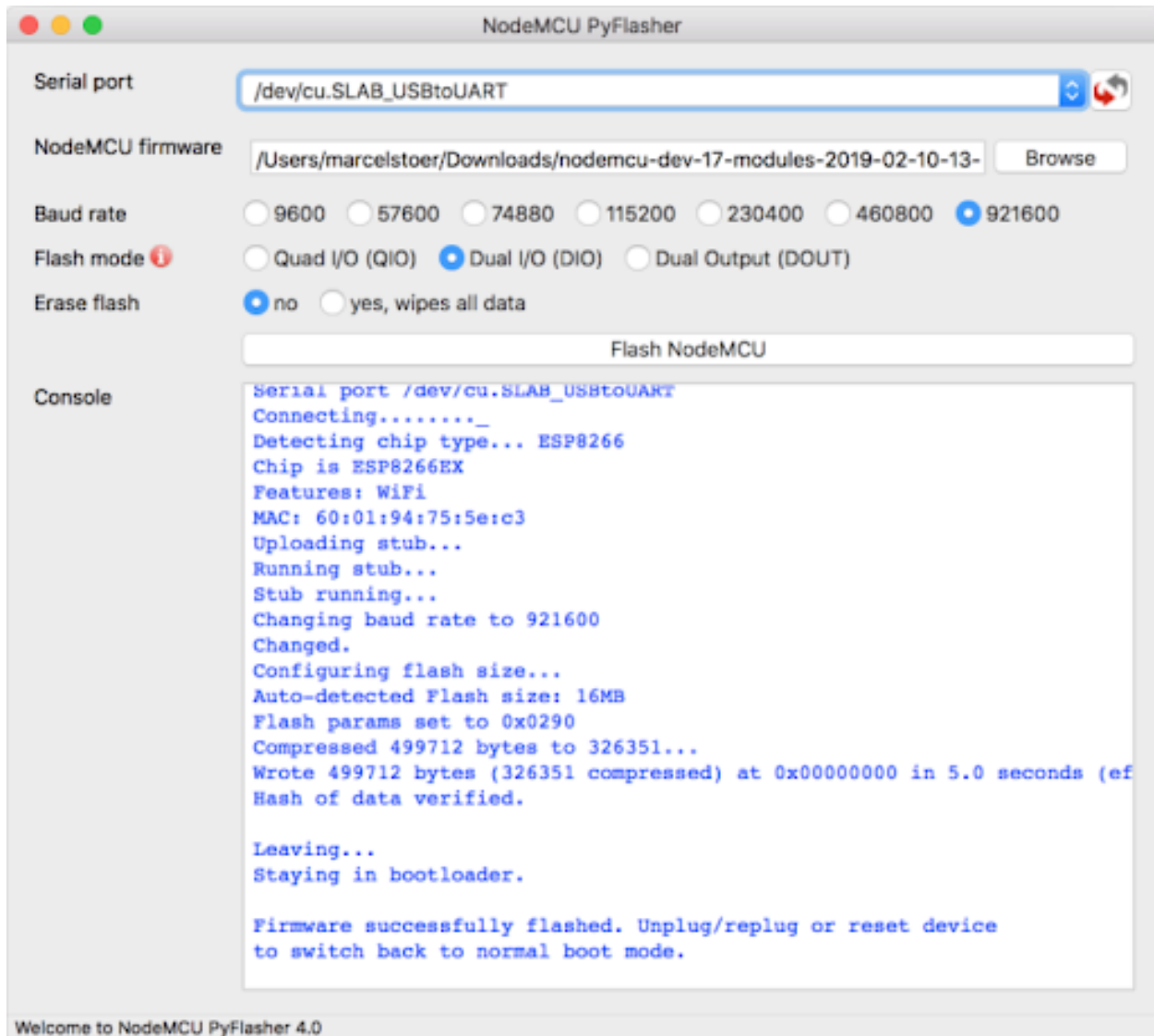
2. Install NodeMCU PyFlasher and flashing Micropython on ESP8266

NodeMCU PyFlasher is a new GUI tool to flash NodeMCU based on esptool.py and wxPython.

First, you have *to connect the ESP8266 to your computer*. Take the USB cable from the kit and Put the USB-C in one of your ports and then connect the micro-USB to the microcontroller.

Check this link [install NodeMCU](#), if you have High Sierra. Scroll down to the executables and click on the third executable (*NodeMCU-PyFlasher-4.0.dmg*).

After the installation will be completed, click on the NodeMCU PyFlasher .exe file and you should have a similar window:



First, you have to choose your serial port, in this case the port is: “/dev/cu.SLAB_USBtoUART”

Succeeding, you have to click on “Browse” and go to the folder where you installed Micropython and choose the .bin file. Next you should choose the *Baud rate* like in the image above

If you encounter errors, you need to reduce the baud rate (for example 1115200 or up down). “921600” is the speed read by serial port.

As final step, you have to click on the button “Flash NodeMCU”.

Congratulations, now you have Micropython on your ESP8266!

General Architecture of Wyliodrin STUDIO

Wyliodrin STUDIO consists of a series of plugins that we used to build the different parts of our application.

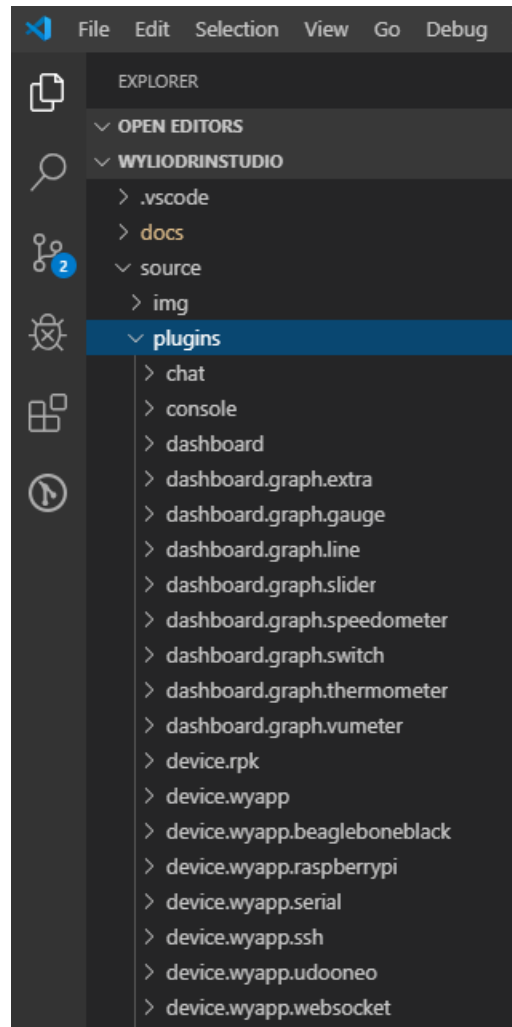
Basically, a plugin is a component of the program that will help you apply different features. Due to the fact that Wyliodrin Studio supports plugins, it enables customization, which means that you will be able contribute to the development and improvement of our application.

To design the user interface we chose the [Vue framework](#) and for data synchronization we used [VueX](#) library, which is deeply integrated into Vue and exploits its reactivity.

3.1 Plugin architecture

Each plugin is a folder in the **source/plugins**.

In order to create your own plugin, you should open the folder that you cloned before with a source-code editor, like **Visual Studio Code**. After that, you will have to open the **plugins** folder, that represents the “storage center” for all the plugins and that is found inside the **source** folder. Here, in **plugins**, you will create a new folder, named after the plugin you’d like to add.



We recommend for the plugin name to be lowercase, and the words separated by “.” For example, we’ll create the **my.new.plugin** folder.

The main components that you’ll need to create for your plugin are:

- The **data** folder: contains a sub-directory, **img**, which can also include different folders that you’ll need in order to keep the images that you use inside your `.vue` files.
- The **style** folder: contains the `.less` files, where we apply the CSS design for the different vue-components.
- The **translations** folder: consists of the `messages-ln.json` files (ln=language abbreviation). More details regarding this subject can be found [here](#).
- The **views** folder, optional, recommended only if you will create `.vue` files to design the user interface for your plugin. (For example, it can contain the file `MyVueFile.vue`)
- The **package.json** file, which contains an object with the primary details regarding your plugin:

Property title	Description	Required / Optional	Default value
<code>name</code>	the name of the plugin (“button.example”)	required	-
<code>version</code>	0.0.1	required	“0.0.1”
<code>main</code>	the main file of the plugin, that will be “index.js”	required	“index.js”
<code>plugin</code>	an object where we specify the characteristics of the plugin	required	-

The properties of the “*plugin*” component are:

Property title	Description	Required / Optional	Default value
<i>consumes</i>	we specify from which other plugins our plugin uses exported functions (required “ <i>workspace</i> ”)	required	[“workspace”]
<i>provides</i>	we specify if our plugin functions will be exported (“ <i>example_button</i> ”)	optional	[]
<i>target</i>	for which version of the program the plugin should be working: browser or electron	required	-

As an example, a *package.json* file should look like this:

```
{
  "name": "my.new.plugin",
  "version": "0.0.1",
  "main": "index.js",
  "private": false,
  "plugin": {
    "consumes": ["workspace"],
    "provides": ["my_new_plugin"],
    "target": ["browser", "electron"]
  }
}
```

- The **index.js** file, which will be your main file.

Here, you can import all the *.vue* files that you need to register.

For example, if you previously create some Vue components to design the user interface, the first line in your *index.js* could look like that:

```
import MyVueFile from './views/MyVueFile.vue';
```

After that, you’ll need to instantiate an object that can be empty, or that can contain different functions that you’ll use.

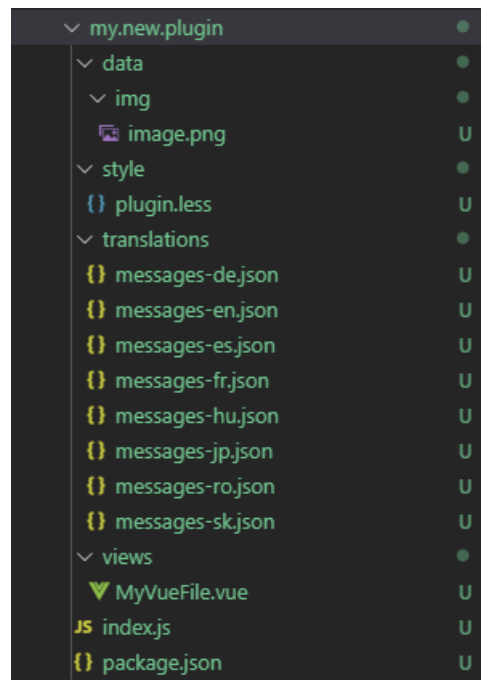
The most important component of this file is the **setup** function that has to be exported, its purpose being to register your plugin and to make it functional inside the application.

```
export function setup(options, imports, register)
{
  /* Collect the functions exported by the consumed plugins */
  studio = imports;

  /* Here goes your code */

  register(null, {});
}
```

At the end, the folder should look like this:



3.2 Dependencies

We are using the **webpack** module to process the Wylidrin STUDIO application. If you're not familiarized with webpack, you should consult the theory presented in their [documentation](#), in order to understand which are the core concepts and how the modules that we use are mapped into the “dependency graph”.

As you probably read before, there are 2 different options to build the code, depending on the version that you are using:

- *Standalone*

```
npx webpack
```

- *Browser*

```
npx webpack --config=webpack.browser.config.js
```

Once the code was built, a folder named “**build**” is created. Its content represents the distribution code, which means a “minimized and optimized output of our build process that will eventually be loaded”. More details can also be found [here](#).

To pack (or “bundle”) a dependency, we need to install the module locally. These dependencies are copied in the *build* folder, but they are not available yet for the browser version of Wylidrin STUDIO.

```
npm install archiver --save
```

We also created the **devDependencies** option, which allow to some particular dependencies to work not only for the electron edition, but also for the browser one. They are saved in the main *package.json* file of the program, as *devDependencies* property, and they are installed using the command:

```
npm install highcharts --save-dev
```

3.2.1 Imports

Each plugin exports in its main file “index.js” a **setup** function, designed to register the plugin. The structure of this function is:

```
export function setup(options, imports, register)
{
    /* the function code */
}
```

As you can see, one of the parameters of this function is **imports**.

The *imports* object has as purpose to collect all the functions and dependencies from the other plugins that our plugin consumes.

For example, let’s suppose that you have a plugin called “*test.plugin*”, which depends on the “workspace” and “projects” plugins. This means that the content of its *package.json* file will be:

```
{
  "name": "test.plugin",
  "version": "0.0.1",
  "main": "index.js",
  "private": false,
  "plugin": {
    "consumes": ["workspace", "projects"],
    "provides": [],
    "target": ["electron", "browser"]
  }
}
```

The fact that your plugin *consumes* these 2 plugins means that the **imports** object will include all their modules and will allow you to access all their functions. Therefore, your *setup* function from the “index.js” file could look like this:

```
let studio = null;

export function setup (options, imports, register)
{
    studio = imports;
```

(continues on next page)

(continued from previous page)

```

    /* use the registerTab function from the workspace plugin */
    studio.workspace.registerTab('TEST_TAB', 100, TestTab, {
        visible ()
        {
            /* use the getCurrentProject function from the projects_
↪plugin to make
            the tab visible only if there is a project opened */

            return !!studio.projects.getCurrentProject();
        }
    });
}

```

3.2.2 Provides

As it was specified in [this](#) section, “**provides**” is a property assigned to the “plugin” property in the *package.json* file of each plugin. The idea around this property is to indicate if a plugin will export its own functions and modules to be used by other plugins.

For example, let’s assume that you have the same plugin, “test.plugin”, which doesn’t provide anything. This means that all its functions will be private and no other plugin will be able to use them, not even if it specifies that it “consumes” your plugin.

In this case, the *package.json* file of your plugin will look like this:

```

{
  "name": "test.plugin",
  "version": "0.0.1",
  "main": "index.js",
  "private": true,
  "plugin": {
    "consumes": [],
    "provides": [],
    "target": ["electron", "browser"]
  }
}

```

And the *index.js* file will look like this:

```

export function setup (options, imports, register)
{
    studio = imports;

    /*Here goes your code*/
    register (null, {});
}

```

But if you want for your plugin to provide all its functions so that the others plugins may access and use them, you have to indicate this option inside the “provides” property. You should be careful at the fact that the provided object should not contain and “.” in its name, unlike the plugin name.

Therefore, the content of the *package.json* should be:

```
{
  "name": "test.plugin",
  "version": "0.0.1",
  "main": "index.js",
  "private": true,
  "plugin": {
    "consumes": [],
    "provides": ["test_plugin"],
    "target": ["electron", "browser"]
  }
}
```

As you can see, your “test.plugin” provides the “test_plugin” object, which means that if another plugin it’s using its functions, it should consume the same “test_plugin” object.

In this situation, the *index.js* file will have the following structure:

```
export function setup (options, imports, register)
{
  studio = imports;

  /* Here goes your code*/

  register (null, {
    test_plugin: test_plugin
  });
}
```

3.3 Architecture Components

3.3.1 Toolbar Buttons

The toolbar is a component located at the top of the window, on which you can add multiple elements.

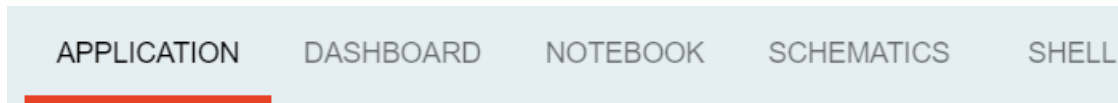


The toolbar buttons are created using the **registerToolBarButton** function. One of the functionalities added in the toolbar using this function is the *Projects Library*, which opens a dialog where the user can manage his applications.



You can learn more about this component [here](#).

3.3.2 Tabs



The tabs are the main components of the workspace, created using the **registerTab** function. They offer the possibility to write and test the code for programming an IoT device, display sensors data, import Fritzing schematics or access the connected device directly through the shell.

The existing tabs at the moment are: **Application**, **Dashboard**, **Notebook**, **Schematics**, **Pin Layout** and **Shell**.

You can find more details about the tabs in [this](#) section.

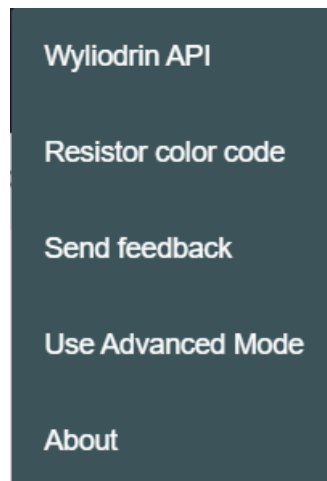
3.3.3 Menu

The Menu is an element created on the toolbar component, represented by the following icon:



When clicked, it opens a menu containing different elements that help the user learn more about Wylodrin STUDIO, send his feedback or switch to the advanced mode.

The components of the menu are:



A better presentation of the menu component and the menu items can be found in [this](#) section.

3.3.4 Connection Button

In the *workspace* plugin we added the connection button, which was designed inside the *DeviceTools.vue* component. It is visible only when there is no device connected to Wylidrin Studio.



On click, it calls the *showConnectionSelectionDialog* and it opens a dialog where the user can see all the available devices. By clicking on a device, he will be asked to input the technical specifications and the login credentials, in order to connect and enable the device functionalities. When the connection was successfully completed, the device status will change from *DISCONNECTED* to *CONNECTED*.

3.3.5 DeviceTool Buttons

These buttons are visible only when a device is connected, because they will replace the *Connection Button*, and they can be different according to the device type.

We added them in the *DeviceTools.vue* component, and this is how they look like:



A better description of this component can be found [here](#).

3.3.6 Status Buttons



The Status Buttons are created with the **registerStatusButton** function. They are used to open the *console* or the *mqtt* server.

The **Console** button opens a console similar to the *shell*.

The **MQTT** button opens an interface where you can choose the port where the *MQTT* server will be opened (publish-subscribe-based messaging protocol).

You can learn more about the status buttons [here](#).

Wylodrin STUDIO enables customization, which means that you may add plugins to extend its features. Plugins may register different components, like buttons specifically designed for devices, workspace tabs, status buttons, toolbar buttons or menus.

Here is a list of plugins of this type, registered at this moment in Wylodrin STUDIO:

4.1 Menu

The menu button is included in the *Menu.vue* component, as a simple image button.



If clicked, it opens a help menu including some topics registered using the **registerMenuItem** function.

registerMenuItem (*name*, *priority*, *action*, *options*)

This function will register a new item in the menu that is displayed in the top left corner of the window. A menu item is a component that will allow the “analysis” of Wylodrin STUDIO, the purpose of the menu being to include details about the application and its operation.

Each item has a *name*, that will be displayed in the menu, a *priority*, which refers to the position of an element in the list of menu items, an *action*, representing the content that will be opened when the item is selected, and additional *options*, that will authorize or will block the user access, depending on their value.

The default value of these options is () => **return true**, which means the menu item will be visible and will allow user access, but it can be customized at the moment of the registration of one element.

If the value of *enabled* will be changed to another function, the name of the menu item will still be visible in the list of all menu items, but it won't permit any user action, because the item will not become usable until the return value of the function will be *true*.

If the value of the *visible* option is changed to another function, the name of the menu item will not appear in the list with all menu items until the return value of the function will become *true*; in this case, when the element is visible, it becomes automatically *enabled*.

Arguments

- **name** (*string*) – the name/id of the menu item
- **priority** (*number*) – the priority of the tab, lower is to the left
- **action** (*function*) – the function to run when clicked
- **options** (*Object*) – additional options, like **visible** or **enabled**; the tab is available for user interaction according to the value of these options

Returns **disposable** – an item that may be disposed

Examples:

```
registerMenuItem('WYLIODRIN_API', 10, () => documentation.openDocumentation());
```

The items currently registered in the menu are:

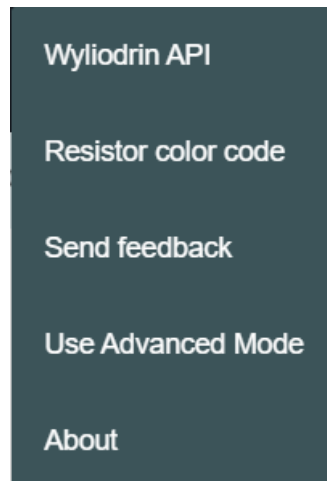
Wyliodrin API: opens a new window with the API documentation

Resistor color code: dialog with the color code of a resistor

Send feedback: dialog where you can write a feedback, having a printscreen attached

Use Advanced/Simple Mode: switch between the simple and advanced (more functionalities included) mode.

About: dialog with a short description of the application



4.2 Toolbar Buttons

These buttons are located in the toolbar, on the top of the main window. A toolbar button is an element that will perform different actions when clicked, according to the component that is relied to it. For example, these buttons may open dialogs that require user inputs.

In order to create this type of buttons, we implemented the **registerToolbarButton** function:

registerToolbarButton (*name*, *priority*, *action*, *iconURL*, *options*)

This function will register a new button in the toolbar.

Each toolbar button has a translatable *name*, that will be displayed under it on mouse hover, a *priority*, which refers to the position of an element in the toolbar buttons list, an *action*, representing the content that will be opened when the button is selected, an *icon* that will represent the actual symbol of the button and on which the user will be able to click, and additional *options*, that will authorize or will block the user access, depending on their value.

The default value of these options is () => **return true**, which means the toolbar button will be visible and will allow user access, but it can be customized at the moment of the registration of one element.

If the value of *enabled* will be changed to another function, the name of the toolbar button will still be visible in the list of all toolbar buttons, but it won't permit any user action, because the button will not become usable until the return value of the function will be *true*.

If the value of the *visible* option is changed to another function, the name of the toolbar button will not appear in the list with all toolbar buttons until the return value of the function will become *true*; in this case, when the element is visible, it becomes automatically *enabled*.

Arguments

- **name** (*string*) – the name/id of the toolbar button
- **priority** (*number*) – the priority of the tab, lower is to the left
- **action** (*function*) – the function to run when clicked
- **iconURL** (*string*) – the relative path to the image assigned
- **options** (*Object*) – additional options, like **visible** or **enabled**; the button is available for user interaction according to the value of these options

Returns **disposable** – an item that may be disposed

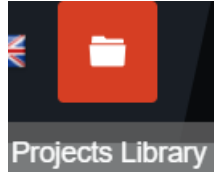
Examples:

```
let time = new Date();

registerToolbarButton('TOOLBAR_BUTTON', 10, () => showNotification('You created a_
↪toolbar button!'), 'plugins/projects/projects/data/img/icons/button.svg', {
  visible() {
    return time.getHours() > 8;
  }
});
```

we register a button having the translation key 'TOOLBAR_BUTTON', the priority 10, that on click will pop up a notification with the content: "You created a toolbar button". We need to specify the relative path to the image related to the button.

This function also modifies the default value of the *visible* additional options, making the button visible for the user only after 8 AM.



4.3 Tabs

The tabs are components of our application and accomplish various functions that help you handling your projects and interacting with the device that is connected to Wylodrin STUDIO.

They are integrated with the **registerTab** function:

registerTab (*name*, *priority*, *component*, *options*)

This function will register a new tab in the workspace.

Each tab has a *title*, that will be displayed in the workspace, a *priority*, which refers to the position of a tab in the list of tabs, a *component*, representing the actual content and functionality of the tab, and additional *options*, that will authorize or will block the user access, depending on their value.

The default value of these options is `() => return true`, which means the menu item will be visible and will allow user access, but it can be customized at the moment of the registration of one element.

If the value of *enabled* will be changed to another function, the name of the menu item will still be visible in the list of all menu items, but it won't permit any user action, because the item will not become usable until the return value of the function will be *true*.

If the value of the *visible* option is changed to another function, the name of the menu item will not appear in the list with all menu items until the return value of the function will become *true*; in this case, when the element is visible, it becomes automatically *enabled*.

Arguments

- **name** (*string*) – the translation ID of the title of the tab
- **priority** (*number*) – the priority of the tab, lower is to the left
- **component** (*Vue*) – the Vue component to display
- **options** (*options*) – additional options, like **visible** or **enabled**; the tab is available for user interaction according to the value of these options;

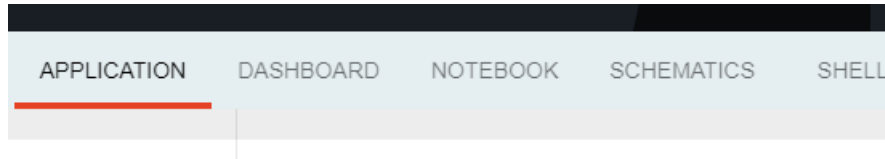
Returns **disposable** – an item that may be disposed { *disposable()* }

Examples:

```
let time = new Date();

registerTab('PROJECT_NOTEBOOK', 300, Notebook, {
  enabled () {
    return time.getHours() > 8;
  }
});
```

A list of the currently existing tabs:



The tabs are registered in the *workspace* plugin. They can be accessed only if their “*enabled*” property is *true*, which means that you have to validate a certain condition: have an opened project or be connected to a device.

4.4 DeviceTool Buttons

These buttons are visible only when a device is connected and they can be different according to the device type.

We added them in the *DeviceTools.vue* component, and this is how they look like:



They were previously registered using the **registerDeviceToolButton** function:

registerDeviceToolButton (*deviceType*, *name*, *priority*, *action*, *iconURL*, *options*)

This function is used to register a new device tool button, specific for every device type.

For example, when a Raspberry Pi board is connected, the following buttons become available: **Run**, **Stop**, **TaskManager**, **PackageManager**, **NetworkManager**.

Each device button require a *deviceType*, to know for which type of device we are registering the customized button, it has a translatable *name*, that will be displayed under it on mouse hover, a *priority*, which refers to the position of an element in the device buttons list, an *action*, representing the content that will be opened when the button is selected, an *icon* that will be the actual symbol of the button and on which the user will be able to click, and additional *options*, that will authorize or will block the user access, depending on their value.

The default value of these options is () => **return true**, which means the device button will be visible and will allow user access, but it can be customized at the moment of the registration of one element.

If the value of *enabled* will be changed to another function, the name of the device button will still be visible in the list of all device buttons, but it won't permit any user action, because the button will not become usable until the return value of the function will be *true*.

If the value of the *visible* option is changed to another function, the name of the device button will not appear in the list with all device buttons until the return value of the function will become *true*; in this case, when the element is visible, it becomes automatically *enabled*.

Arguments

- **deviceType** (*string*) – the device driver type the button is for
- **name** (*string*) – the name/id of the menu item
- **priority** (*number*) – the priority of the tab, lower is to the left

- **action** (*function*) – the function to run when clicked
- **iconURL** (*string*) – the relative path to the image assigned
- **options** (*Object*) – additional options, like **visible** or **enabled**; the button is available for user interaction according to the value of these options

Returns **disposable** – an item that may be disposed

Examples:

```
let time = new Date();

registerDeviceToolButton('RUN', 10, => showNotification('You clicked the Run_
↔button!'),
    'plugins/studio/workspace/data/img/icons/button.svg', {
    visible() {
        return time.getHours() > 8;
    }
});
```

Here, we registered a device tool button having the translation key 'DEVICETOOL_BUTTON', the priority 10, that on click will pop up a notification with the content: "You created a device tool button!".

The button will be visible for an user only after 8 AM.

4.5 Status Buttons

The last component of the workspace is represented by the status buttons: **Console** and **MQTT**. A status button is an element that will perform different actions when clicked, according to the component that is relied to it. For example, these buttons may open terminals or interfaces that require user inputs.

They are created using the **registerStatusButton** function.



registerStatusButton (*name, priority, component, iconURL, options*)

This function will register a new button in the status bar that is displayed in the bottom of the window.

Each status button has a translatable *name*, that will be displayed under it on mouse hover, a *priority*, which refers to the position of an element in the status buttons list, a *component*, representing the content that will be shown when the button is clicked, an *icon* that will represent the actual symbol of the button and on which the user will be able to click, and additional *options*, that will authorize or will block the user access, depending on their value.

The default value of these options is `() => return true`, which means the status button will be visible and will allow user access, but it can be customized at the moment of the registration of one element.

If the value of *enabled* will be changed to another function, the name of the status button will still be visible in the list of all status buttons, but it won't permit any user action, because the button will not become usable until the return value of the function will be *true*.

If the value of the *visible* option is changed to another function, the name of the status button will not appear in the list with all status buttons until the return value of the function will become *true*; in this case, when the element is visible, it becomes automatically *enabled*.

The *statusButtons* registered at the moment can open the **Console** and the **Mqtt** server interface.

Arguments

- **name** (*string*) – the name/id of the menu item
- **priority** (*number*) – the priority of the tab, lower is to the left
- **component** (*Vue*) – the Vue component to display
- **iconURL** (*string*) – the relative path to the image assigned
- **options** (*Object*) – additional options, like **visible** or **enabled**; the button is available for user interaction according to the value of these options

Returns **disposable** – - an item that may be disposed

Examples:

```
registerStatusButton('CONSOLE', 1, Console, 'plugins/studio/console/data/img/
↪icons/terminal-icon.svg');
```

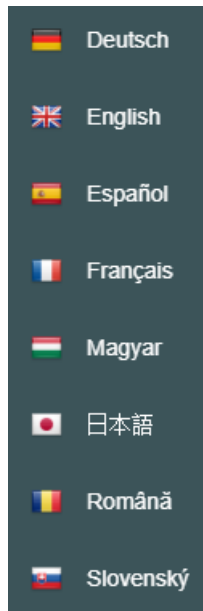
The **Console** button opens a console similar to the *shell*, while the **MQTT** button opens an interface where you can choose the port where the *MQTT* server will be opened (the default port is 1883). MQTT is a publish-subscribe-based messaging protocol.

4.6 Language

The language button is included in the *LanguageMenu.vue* component and its corresponding image, a flag, changes dynamically according to the selected language.



Here's a list with all the languages available at this moment:



When a language is selected from the list, the **setLanguage** function is called, which is using the [internationalization \(i18n\)](#) process, and the new language is updated, meaning that all the keys will be translated. More details about the translation function are discussed [here](#).

Deploy Application

Broadly speaking, software deployment consists of all the porcesses required for preparing the a software application to run and operate in a specific environment. It involves installation, configuration, testing and making changes to optimize the performance of the software. In Wylodrin Studio, **deploying an application** basically means to put it into production, after its prototype is finished and tested.

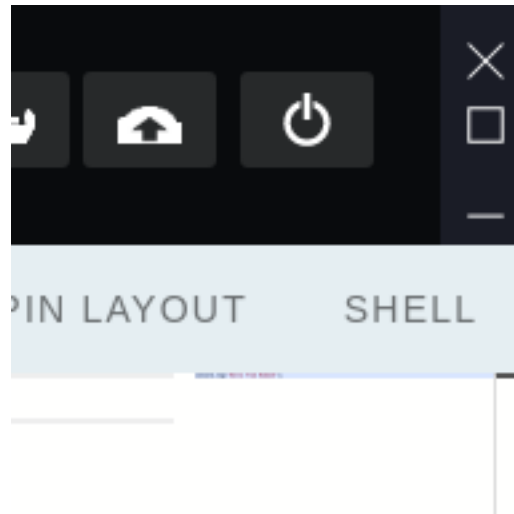
For the moment Wylodrin Studio provides support just for **Docker**. The application will be deployed in a container, which is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably. In this way the app will be able to start automatically and run in the background.

This feature contains two main parts.

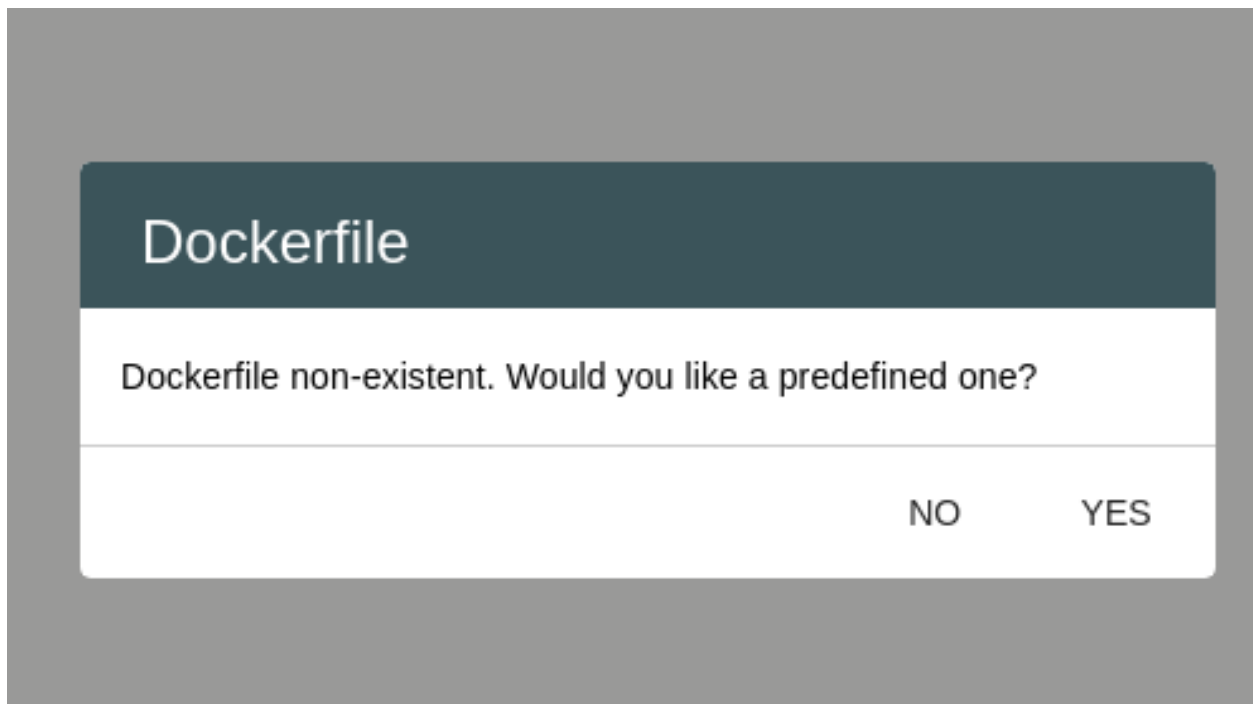
5.1 Deploy an Application

5.1.1 Start a deployment

To deploy a project you have to use the **Deploy** button.



After pressing it, the **Dockerfile** pop-up will appear.



This means that you do not have any dockerfile in the folder of your project. You now have the possibility to create a dockerfile through **Wyliodrin Studio**, or to make one of your own.

5.1.2 Setup the deployment

You will be able to customize your dockerfile in the **Deployment Settings** pop-up.

In this dialog, you have multiple options to customize your dockerfile. The process options are:

Interactive	the container is running in the foreground and has the console attached
Service	the container is running in detached mode

The restart options are:

Do not restart	do not automatically restart the container when it exits
Always	restart the container always, regardless of the exit status
On failure	restart the container only when it exits with a non-zero status
Unless stopped	always restart the container, except if it was put into stopped state before the Docker daemon was stopped.

The network options are:

Private	use docker's default networking setup
Same as device	use the host's network stack inside the container

Other options are:

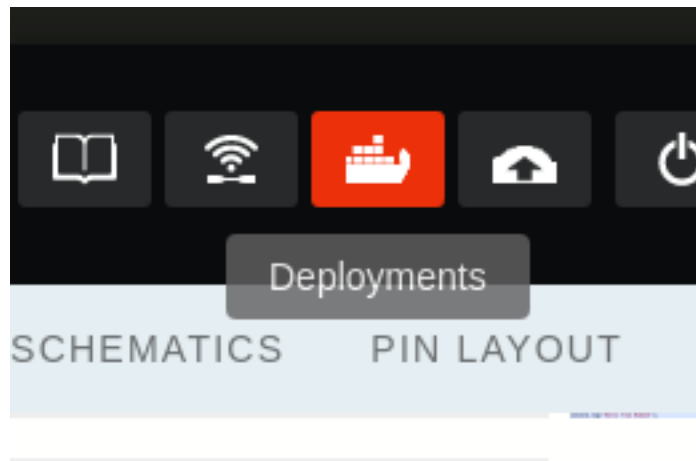
Remove container at exit	automatically clean up the container and remove the file system when the container exits
Priviledged container	give extended privileges and acces to all devices to the container

As you can see, there are already some default options set. However, you can always change them or add other options that you need in the **Additional Options** field.

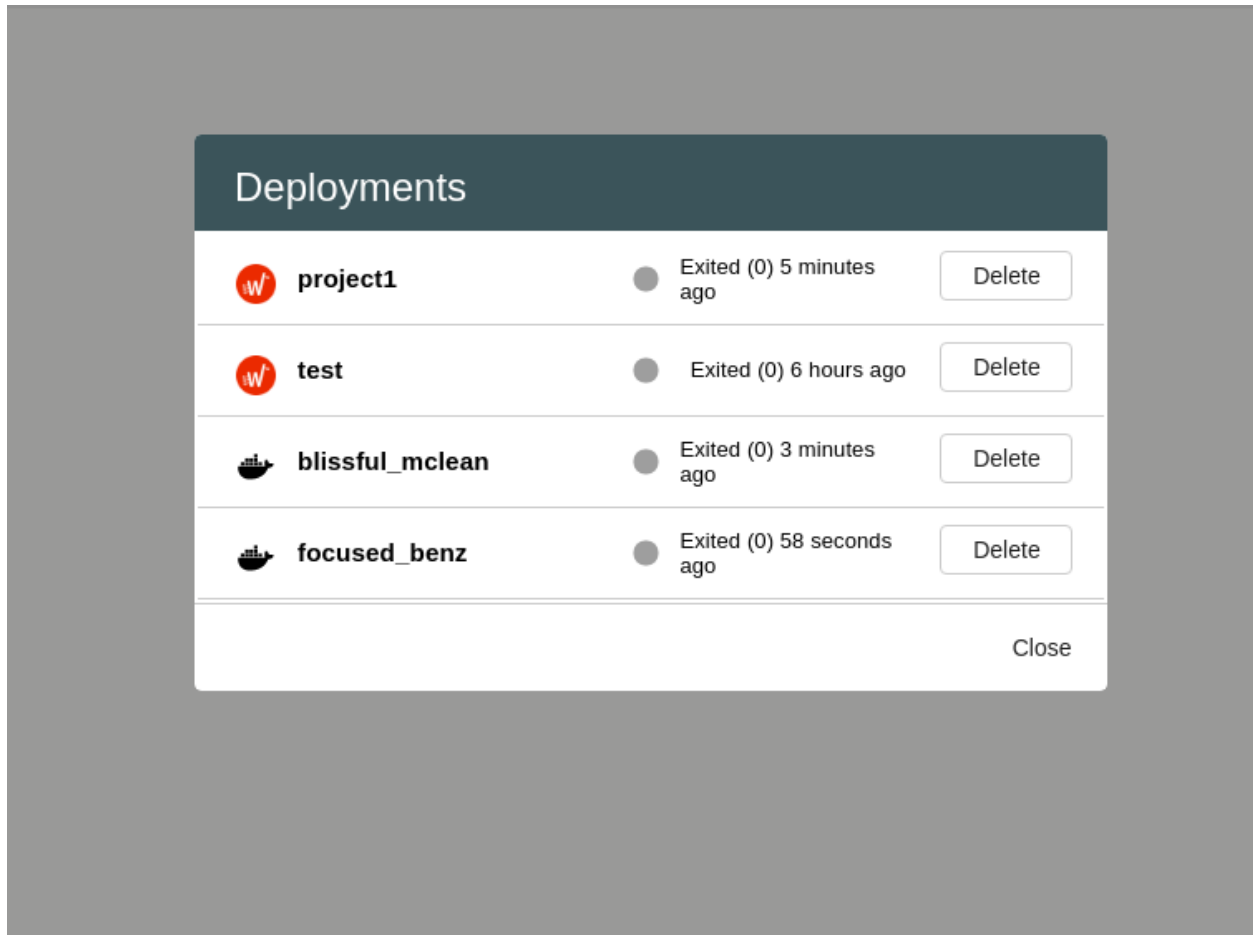
By doing these actions, you succesfully deployed your project in a container. This container has the same name as your project and can be found in the **Deployments** pop-up.

5.2 Manage Deployments

Wylodrin STUDIO allows you to manage your deployed apps. By pressing the **Deployments** button you can manage both your containers created in **Wylodrin Studio**, and the ones created locally on your machine.



After pressing this button you will be shown a list of all the containers.



In the list you will find two types of containers. The ones with the **Wyliodrin Studio** logo are the one created inside the application, whilst the ones with the docker icon are created locally. You may also see in which state the container can be found in that particular moment, as well as delete or stop the containers.

6.1 Workspace plugin API

“Workspace” is the main plugin in our application. It exports the “*workspace*” object, containing a series of functions that we use in every other plugin.

6.1.1 Data Types

class Device()

Device Identification

Arguments

- **id** (*String*) – unique id for the device (determined by the driver)
- **name** (*String*) – name of the device
- **type** (*String*) – type of the device (the device type id that reported the device)

disposable()

a function that is called when the item may be deleted

6.1.2 Tabs

registerTab (*name*, *priority*, *component*, *options*)

This function will register a new tab in the workspace.

Each tab has a *title*, that will be displayed in the workspace, a *priority*, which refers to the position of a tab in the list of tabs, a *component*, representing the actual content and functionality of the tab, and additional *options*, that will authorize or will block the user access, depending on their value.

The default value of these options is `() => return true`, which means the menu item will be visible and will allow user access, but it can be customized at the moment of the registration of one element.

If the value of *enabled* will be changed to another function, the name of the menu item will still be visible in the list of all menu items, but it won't permit any user action, because the item will not become usable until the return value of the function will be *true*.

If the value of the *visible* option is changed to another function, the name of the menu item will not appear in the list with all menu items until the return value of the function will become *true*; in this case, when the element is visible, it becomes automatically *enabled*.

Arguments

- **name** (*string*) – the translation ID of the title of the tab
- **priority** (*number*) – the priority of the tab, lower is to the left
- **component** (*Vue*) – the Vue component to display
- **options** (*options*) – additional options, like **visible** or **enabled**; the tab is available for user interaction according to the value of these options;

Returns **disposable** – an item that may be disposed { *disposable()* }

Examples:

```
let time = new Date();

registerTab('PROJECT_NOTEBOOK', 300, Notebook, {
  enabled () {
    return time.getHours() > 8;
  }
});
```

In this example, the Notebook tab will be enabled and will allow user access only after 8AM. Until then, it will appear in the list of tabs as it follows:



registerMenuItem (*name*, *priority*, *action*, *options*)

This function will register a new item in the menu that is displayed in the top left corner of the window. A menu item is a component that will allow the “analysis” of Wylodrin STUDIO, the purpose of the menu being to include details about the application and its operation.

Each item has a *name*, that will be displayed in the menu, a *priority*, which refers to the position of an element in the list of menu items, an *action*, representing the content that will be opened when the item is selected, and additional *options*, that will authorize or will block the user access, depending on their value.

The default value of these options is () => **return true**, which means the menu item will be visible and will allow user access, but it can be customized at the moment of the registration of one element.

If the value of *enabled* will be changed to another function, the name of the menu item will still be visible in the list of all menu items, but it won't permit any user action, because the item will not become usable until the return value of the function will be *true*.

If the value of the *visible* option is changed to another function, the name of the menu item will not appear in the list with all menu items until the return value of the function will become *true*; in this case, when the element is visible, it becomes automatically *enabled*.

Arguments

- **name** (*string*) – the name/id of the menu item
- **priority** (*number*) – the priority of the tab, lower is to the left
- **action** (*function*) – the function to run when clicked
- **options** (*Object*) – additional options, like **visible** or **enabled**; the tab is available for user interaction according to the value of these options

Returns **disposable** – - an item that may be disposed

Examples:

```
registerMenuItem('WYLIODRIN_API', 10, () => documentation.openDocumentation());
```

In this example, the *Wylodrin API* menu element will open a new documentation window when clicked.

renameMenuItem (*prevName*, *actualName*)

Rename an item from the menu.

The previous parameters that were set for the current menu item will remain unchanged, but the name of the element will be updated in the list of menu items.

Arguments

- **prevName** (*string*) – the initial name of the item
- **actualName** (*string*) – the actual name of the item

Returns **disposable** – - an item that may be disposed

Examples:

```
renameMenuItem('WYLIODRIN_API', 'WYLIODRIN_STUDIO_API');
```

registerDeviceToolButton (*deviceType*, *name*, *priority*, *action*, *iconURL*, *options*)

This function is used to register a new device tool button, specific for every device type.

For example, when a Raspberry Pi board is connected, the following buttons become available: **Run**, **Stop**, **TaskManager**, **PackageManager**, **NetworkManager**.

Each device button require a *deviceType*, to know for which type of device we are registering the customized button, it has a translatable *name*, that will be displayed under it on mouse hover, a *priority*, which refers to the position of an element in the device buttons list, an *action*, representing the content that will be opened when the button is selected, an *icon* that will be the actual symbol of the button and on which the user will be able to click, and additional *options*, that will authorize or will block the user access, depending on their value.

The default value of these options is () => **return true**, which means the device button will be visible and will allow user access, but it can be customized at the moment of the registration of one element.

If the value of *enabled* will be changed to another function, the name of the device button will still be visible in the list of all device buttons, but it won't permit any user action, because the button will not become usable until the return value of the function will be *true*.

If the value of the *visible* option is changed to another function, the name of the device button will not appear in the list with all device buttons until the return value of the function will become *true*; in this case, when the element is visible, it becomes automatically *enabled*.

Arguments

- **deviceType** (*string*) – the device driver type the button is for
- **name** (*string*) – the name/id of the menu item
- **priority** (*number*) – the priority of the tab, lower is to the left
- **action** (*function*) – the function to run when clicked
- **iconURL** (*string*) – the relative path to the image assigned
- **options** (*Object*) – additional options, like **visible** or **enabled**; the button is available for user interaction according to the value of these options

Returns **disposable** – - an item that may be disposed

Examples:

```
let time = new Date();

registerDeviceToolButton('RUN', 10, => showNotification('You clicked the Run_
↪button!'),
    'plugins/studio/workspace/data/img/icons/button.svg', {
    visible() {
        return time.getHours() > 8;
    }
});
```

This example creates a characteristic device button called 'Run', that will display a notification when clicked. Because of the *visible* option, the button will appear in the list of all tool buttons only after 8AM.

6.1.3 Status Bar

registerStatusButton (*name*, *priority*, *component*, *iconURL*, *options*)

This function will register a new button in the status bar that is displayed in the bottom of the window.

Each status button has a translatable *name*, that will be displayed under it on mouse hover, a *priority*, which refers to the position of an element in the status buttons list, a *component*, representing the content that will be shown when the button is clicked, an *icon* that will represent the actual symbol of the button and on which the user will be able to click, and additional *options*, that will authorize or will block the user access, depending on their value.

The default value of these options is `() => return true`, which means the status button will be visible and will allow user access, but it can be customized at the moment of the registration of one element.

If the value of *enabled* will be changed to another function, the name of the status button will still be visible in the list of all status buttons, but it won't permit any user action, because the button will not become usable until the return value of the function will be *true*.

If the value of the *visible* option is changed to another function, the name of the status button will not appear in the list with all status buttons until the return value of the function will become *true*; in this case, when the element is visible, it becomes automatically *enabled*.

The *statusButtons* registered at the moment can open the **Console** and the **Mqtt** server interface.

Arguments

- **name** (*string*) – the name/id of the menu item
- **priority** (*number*) – the priority of the tab, lower is to the left
- **component** (*Vue*) – the Vue component to display
- **iconURL** (*string*) – the relative path to the image assigned
- **options** (*Object*) – additional options, like **visible** or **enabled**; the button is available for user interaction according to the value of these options

Returns **disposable** – an item that may be disposed

Examples:

```
registerStatusButton('CONSOLE', 1, Console, 'plugins/studio/console/data/img/
↪icons/terminal-icon.svg');
```

In this example, a new status button is created. The *Console* component has to be previously created as a Vue component and imported in the *index.js* file where the new status button is registered:

```
import Console from './views/Console.vue';
```

openStatusButton (*name*)

Open a status button, using the **dispatchToStore** function to send to the *activeStatusButton* variable from the workspace store the value of the chosen status button.

Arguments

- **name** (*string*) – the name of the status button to open

Examples:

```
openStatusButton('CONSOLE');
```

closeStatusButton()

Close a status button, using the **dispatchToStore** function to send to the *activeStatusButton* variable from the workspace store an empty string, which means that the currently open status button is no longer available.

Examples:

```
closeStatusButton();
```

6.1.4 Data Store

registerStore (*namespace, store*)

This function registers a new namespaced store.

A “*store*” is basically a container that holds the application state. Since a Vuex store is reactive, when a Vue component needs or changes a variable state, it will reactively and efficiently update the values.

Arguments

- **namespace** (*string*) – the name/id of the menu item
- **store** (*Object*) – the actual store object, imported from the ‘*store.js*’ file of the plugin

Returns *undefined* –

Examples:

```
registerStore('projects', projectStore);
```

getFromStore (*namespace, variable*)

Gets the value of a variable from a selected store.

Arguments

- **namespace** (*string*) – the name of the store where the variable is registered

- **variable** (*string*) – the name of the variable to process

Examples:

```
let project = getFromStore('projects', 'currentProject');
```

dispatchToStore (*namespace, action, data*)

Sends data to a selected store promptly and updates the state and value of a certain variable.

Arguments

- **namespace** (*string*) – the name of the store where the data will be dispatched
- **action** (*string*) – the variable to be updated
- **data** (*Object*) – additional data to send to the variable

Examples:

```
dispatchToStore('projects', 'currentProject', null);
```

6.1.5 Vue

registerComponent (*component*)

Register a Vue component.

Arguments

- **component** (*Vue*) – the Vue component to be registered

Examples:

```
registerComponent(MyComponent);
```

setWorkspaceTitle ()

This function sets the title of the workspace according to the name of the current project.

The workspace title will be displayed to the left of the tabs list.

Examples:

```
setWorkspaceTitle (project.name);
```

6.1.6 Device Drivers

registerDeviceDriver (*name*, *deviceDriver*)

This function registers a new device type. It requires a *name* that indicates the type of the device for which it will register the driver, and the actual device driver object, that include a series of properties and functions.

Arguments

- **name** (*string*) – device type name
- **deviceDriver** (*DeviceDriver*) – actual device driver, consists of a series of functions necessary to represent, connect, disconnect or set up a device.

Examples:

```
registerDeviceDriver('my_device', deviceDriver);
```

updateDevices (*type*, *dev*)

This function updates the list of devices for a device type. It's required to know the *type* of the device that will be updated, and the list with all the devices that will be attached to the selected type of device.

Arguments

- **type** (*string*) – device type, has to be registered
- **dev** (*Array.<Device>*) – a list of devices (*Device()*)

Examples:

```
updateDevices (myDevices);
```

connect (*device*, *options*)

The purpose of this function is to connect Wyliodrin STUDIO to a device.

In order to connect, it's required to have a valid device object and the corresponding connection options. This process demands to constantly check the device status.

The statuses that a device can have are:

DISCONNECTED - this is offline

CONNECTING - trying to connect

SYNCHRONIZING - trying to synchronize with the device

CONNECTED - this is online

ISSUE - there is some issue, the system is partially functional

ERROR - there is an error with the system

Arguments

- **device** (*Device*) – the device to connect to
- **options** (*Device*) – connect options

getDevice ()

Returns a device from the store.

This function has no parameters and it's using the **getFromStore** function, which returns a **device** object, with all its properties. It's useful to work with it each time you want to manipulate the currently connected device and you need to know its type.

Examples:

```
let device = getDevice ();
```

getStatus ()

This function returns the status of a device.

The function has no parameters and calls the **getFromStore** function, which returns from the workspace store a string representing the current status of the device the user tries to work with.

Examples:

```
let status = getStatus();
```

disconnect ()

This function disconnects the currently connected device from Wyliodrin STUDIO, which means that it deletes the connections and characteristics of this device, as reported by the type of disconnection that the user chooses:

StandBy -

Disconnect -

Turn Off -

showConnectionSelectionDialog()

This function opens a customized dialog box, used to select a device that will be connected to Wyliodrin Studio.

It's called when the user clicks on the 'Connect' button and it shows a dialog containing a list with all the devices available for connection.

6.2 Projects plugin API

The “**projects**” plugin is the second most important component in our application. Same as “*workspace*”, it has its own store, where we register the applications the user creates, in order to manage properly his activity.

6.2.1 Data Types

class Project()

Project Identification

Arguments

- **date** (*string*) – date and time of the last time the project was accessed
- **folder** (*string*) – absolute path to the project
- **language** (*string*) – programming language of the project
- **name** (*string*) – the actual name of the project

class Language()

Programming Language Identification

Arguments

- **addons** (*object*) – the specific features of the language // language addons
- **icon** (*string*) – path to the language image
- **id** (*string*) – language name
- **fileIcons** (*array*) – array of language specific fileIcons
- **options** (*Object*) – language functions

class file()

File Identification

Arguments

- **file** (*string*) – extension if it's a file
- **children** (*Array.<file>*) – children if it's a folder
- **path** (*string*) – path to object
- **name** (*string*) – name of object

disposable()

a function that is called when the item may be deleted

6.2.2 Programming Languages

`projects.getLanguage(languageID)`

This function returns a programming language object with the following properties: id, title, icons, addons and options.

It requires the unique id that identifies the language in the list of all programming languages.

Arguments

- **languageID** (*string*) – the id of said language

Returns **Language** – - programming language properties

registerLanguage (*id, title, projectIcon, logoIcon, fileTreeIcon, fileIcons, options*)

This function registers a language object by updating the list of all languages with a new programming language having its own specifications and functions.

Every new language has an *id*, its unique identifier, a *title*, which is the actual name of the programming language, a characteristic *icon*, and its own *options* required in order to be working properly.

The accepted languages are: *javascript*, *python*, *bash* and *visual*.

Arguments

- **id** (*string*) – language id
- **title** (*string*) – language title
- **projectIcon** (*string*) – icon that appears in the projects library
- **logoIcon** (*string*) – icon that appears in the new application popup
- **fileTreeIcon** (*string*) – icon that appears at the top of the file tree
- **fileIcons** (*string*) – icons for files
- **options** (*Object*) – language options

Examples:

```
registerLanguage('python', 'Python', 'plugins/languages/python/data/img/project_
↪python.png', 'plugins/languages/python/data/img/python.png', python);
```

In this example, the last parameter, *python* is an object previously created, its properties being the characteristic functions for this programming language.

registerLanguageAddon (*language*, *types*, *boards*, *options*)

This function is used to add an addon to an already existing language. In this case, an addon refers to a specific feature that can be set for a board.

Each addon requires the programming *language* unique id, the type of the *board* for which the feature will be set, the *type* of the actual addon, and the additional functioning options of the feature.

Arguments

- **language** (*Object*) – language id
- **types** (*string*|*Array.<string>*) – addon type
- **boards** (*string*|*Array.<string>*) – addon board
- **options** (*Object*) – addon options

Returns **boolean** – - true if successful, false otherwise

registerEditor (*name*, *languages*, *component*, *options*)

This function registers a new type of editor.

The editor has a *name*, which is a translatable string that will be displayed as the title of the editor, *languages*, which represent the array with all the supported programming languages id's or file extensions, and a *Vue component*, representing the actual content and design of the editor tab.

Arguments

- **name** (*string*) – the name/id of the editor

- **languages** (*Array.<string>*) – the editor languages
- **component** (*Vue*) – the component to display
- **options** (*array*) – the editor options

Returns **boolean** – - true if successful, false otherwise

Examples:

```
registerEditor('EDITOR_ACE', ['py', 'js'], Ace);
```

This is an example of how you can register an Ace editor that will accept python(*py*) and javascript(*js*) programming languages. The *Ace* component will be designed as a Vue component for the editor and imported inside the main file where the new editor is registered.

languageSpecificOption (*project, option*)

This function returns a specific option that was set to a programming language.

In order to obtain it, is required to have the *project* for which the option was set and the actual *name* of the specific option.

Arguments

- **project** (*Project*) – project object
- **option** (*string*) – option

Returns **Object** – the specific option of the programming language

Examples:

```
let sourceLanguage = languageSpecificOption (project, {...});
```

6.2.3 Projects

createEmptyProject (*name, language*)

This function creates a new empty project.

Each project requires a name, that will be entered by the user as a text in the input area, and a programming *language* that the project will use, also chosen by the user.

Arguments

- **name** (*string*) – Project name
- **language** (*string*) – Project language

Returns **Project** – - Project object

Examples:

```
project = createEmptyProject('MyProject', 'py')
```

deleteProject (*project*)

This function deletes all the files related to the project chosen by the user, when he clicks on the “Delete” button. After removing all the files, the *currentProject* and *currentFile* are dispatched to the projects store as *null*.

Arguments

- **project** (*Project*) – Project object

Returns **boolean** – true if succsesful, false otherwise

Examples:

```
deleteProject('MyProject');
```

renameProject (*project, newName*)

This function renames a selected project, when the user clicks on the *Rename* button.

It’s required to know the *project* that will be renamed and the *new name*, that will be entered by the user in the input text area.

Arguments

- **project** (*Project*) – Project object
- **newName** (*string*) – New project name

Returns **boolean** – true if succsesful, false otherwise

Examples:

```
renameProject('MyProject', 'MyRenamedProject');
```

cloneProject (*project, newName*)

This function is used to clone a project, by creating a duplicate of the selected project and assigning to it the “**newName**” value, chosen by the user.

Arguments

- **project** (*Project*) – Project object
- **newName** (*string*) – Cloned project name

Returns **boolean** – true if succsesful, false otherwise

Examples:

```
cloneProject(project, 'MyClonedProject');
```

importProject (*project, data, extension*)

This function imports a project archive.

Loads a new project tree from the user's computer. The archive extension can be ".zip", ".tar" (in this case the files will be extracted), or ".wylloapp" (we are creating recursively the project folder).

Arguments

- **project** (*Project*) – project object
- **data** (*Project*) – data from project
- **extension** (*string*) – archive extension (.zip/.tar/.wylloapp)

Returns **boolean** – true if succsesful, false otherwise

Examples:

```
importProject(project, projectData, '.zip');
```

recursiveCreating (*necessary*)

Recursively generate the project tree structure with paths and names

necessary.item - file item

necessary.item.isdir - is or not directory

necessary.item.children - only if it's a directory

necessary.item.name - name

necessary.item.content - file content only if it's a file

Arguments

- **necessary** (*Object*) – Object representing the details about every file withing the project

Returns **boolean** – true if succsesful, false otherwise

exportProject (*project*)

This function exports a project archive.

It's required to know the project that the user will export, including all of its files and folders, and the path to where the project will be saved in the user's computer. The archive will have the .zip extension.

Arguments

- **project** (*Project*) – project object

Examples:

```
exportProject (project) ;
```

recursiveGeneration (*project, file*)

Recursively generate a deep object with all the contents of a project

Arguments

- **project** (*Project*) – Project object
- **file** (*file*) – File object

Returns *file* – the root of the folder with all its contents

loadProjects ()

Load existing projects.

This function has no parameters. It creates a list with all the existing projects when it's called, by reading all the folders from the main path, *workspacePath*.

Returns *Array.<Project>* -- a list of projects

Examples:

```
let projects = loadProjects();
```

selectCurrentProject (*project*)

This function selects a project from the list with all the projects, when the users clicks on it, and it displays the content of the *project* in the Application tab.

Arguments

- **project** (*Project*) – project object

Returns *boolean* – true if succsesful, false otherwise

loadPreviousSelectedCurrentProject ()

Load a previous selected project. The function has no params, loads the project from local files.

Examples:

```
let project = loadPreviousSelectedCurrentProject();
```

generateStructure (*project*, *isRoot*)

This function generates the tree structure of a project.

Arguments

- **project** (*Project*) – project object
- **isRoot** (*boolean*) – true

Returns *file* – the tree structure

getCurrentProject ()

Get the current project structure.

The **getFromStore** function is called to load the content of the *currentProject* variable from the projects store.

Returns *Project* – project object

6.2.4 Files and Folders

newFile (*project*, *name*, *data*)

This function creates a new file inside a project. For this, it is required that we know the *project* for which the new file is generated, the *name* that the file will have (actually represented by the absolute path to where the file will be created), and, if necessary, the information that will be written in the file.

This option is valid only in the *Advanced Mode*.

Arguments

- **project** (*Project*) – project object
- **name** (*string*) – path to where to create the file
- **data** (*string*) – data to be written to file

Returns *boolean* – true if successful, false otherwise

Examples:

```
newFile(project, '/main.js', 'console.log(\'Hello from JavaScript\');');
```

deleteFile (*project*, *pathTo*)

This function is used to delete a file from a project, and it needs the *project* containing the selected file and the *path* to that file.

Arguments

- **project** (*Project*) – project object
- **pathTo** (*string*) – path to the file

Returns **boolean** – true if succsesful, false otherwise

Examples:

```
deleteFile(project, '/folder/file');
```

saveFile (*project*, *name*, *buffer*)

The purpose of this function is to save a file. It requires the *project* in which the file resides, the *name* of the file, actually represented as the path to the file, and a *buffer* containing the data that will be saved in the created file.

Arguments

- **project** (*Project*) – project object
- **name** (*string*) – path to file
- **buffer** (*string*) – file buffer to be saved

Returns **boolean** – - true if successful, false otherwise

Examples:

```
saveFile(project, '/folder/file', Buffer.from ('...'));
```

loadFile (*project*, *name*)

This function loads the content of a file that was previously saved. In order to open the file, it's needed to know the *project* that the file belongs to, and the full *name* of the file, meaning its path.

Arguments

- **project** (*Project*) – project object
- **name** (*string*) – full file name with path

Returns **Object** – - file content

Examples:

```
let fileContent = loadFile(project, 'FileName');
```

changeFile (*project*, *name*)

Changes the current file to another one.

Arguments

- **project** (*Project*) – project object
- **name** (*string*) – path to file

saveSpecialFile (*project*, *name*, *content*)

The purpose of this function is to save a special settings file and it requires the *project* corresponding to the file, the *name* of the file, actually represented as the path to the file, and the *content* that will be saved in the special settings file.

Arguments

- **project** (*Project*) – project object
- **name** (*string*) – the path to the file
- **content** (*Buffer*) – the content of the file

Returns **boolean** – - true if successful, false otherwise

Examples:

```
saveSpecialFile(project, 'SpecialFileName', Buffer.from (...));
```

loadSpecialFile (*project*, *name*)

This function loads the content of a special settings file that was previously saved. In order to open the file, it's needed to know the *project* that the file belongs to, and the full *name* of the file, meaning its path.

Arguments

- **project** (*Project*) – project object
- **name** (*string*) – the path to the file

Returns **Buffer** – - the content of the special settings file, null otherwise

Examples:

```
loadSpecialFile('MyNewProject', 'SpecialFileName');
```

getDefaultFileName (*project*)

The purpose of this function is to obtain the default file name of a *project*.

Usually, the name of this file is 'main.ext', where *ext* is the extension corresponding to the programming language that defines the project.

Arguments

- **project** (*Project*) – project object

Returns *string* – - name of the default file

getDefaultRunFileName (*project*)

Get the default run file name of a *project*.

Usually, the name of this file is 'main.ext', where *ext* is the extension corresponding to the programming language that defines the project.

Arguments

- **project** (*Project*) – project object

Returns *string* – - name of the default run file

getMakefile (*project*)

This function's purpose is to get the makefile for file name of a *project*.

Arguments

- **project** (*Project*) – project object

Returns *string* – - name of the makefile

getFileCode (*project*, *path*)

This functions returns the code that was written into a file and it needs the *project* where the file is saved and the *path* to the file.

Arguments

- **project** (*Project*) – project object
- **path** (*string*) – the path to the file

Returns *Object* – - the current file code

getCurrentFileCode()

Similar to the one defined before, this function also returns the code, but this time from the current file that is opened in the current project.

Returns **Object** – - the current file code

newFolder (*project*, *name*)

This function creates a new folder inside a project. For this, it is required that we know the *project* for which the new folder is generated and the *name* that the folder will have. The name is actually represented by the absolute path to where the folder will be created.

This option is valid only in the *Advanced Mode*.

Arguments

- **project** (*Project*) – Project object
- **name** (*string*) – path to where to create the folder

Returns **boolean** – true if succsesful, false otherwise

Examples:

```
newFolder(project, '/folder/folder2');
```

deleteFolder (*project*, *pathTo*)

This function is used to delete a folder from a project, and it needs the *project* containing the selected folder and the *path* to that folder.

Arguments

- **project** (*Project*) – project object
- **pathTo** (*string*) – path to the folder

Returns **boolean** – true if succsesful, false otherwise

Examples:

```
deleteFolder(project, '/folder/folder2');
```

renameObject (*project*, *newName*, *pathTo*)

This function is used to rename a file or a folder included in the currently open project.

It's required to know the *project* for which the change is made, the new *name* that will correspond to the selected object and the path to the file/folder to be renamed.

Available only for the *Advanced Mode*, this function is called when the user chooses the *Rename* option in the menu that shows up by right clicking on a folder/file.

Arguments

- **project** (*Project*) – project object
- **newName** (*string*) – new name
- **pathTo** (*string*) – path to existing file/folder

Returns **boolean** – true if successful, false otherwise

Examples:

```
renameObject(project, 'ObjectNewName', '/folder/file');
```

6.3 Dashboard Graphs Plugins

The purpose of the dashboard plugins is to create a collection of graphs that update their values according to the signals received from a connected device.

The main plugin, “*dashboard*”, designs the Dashboard tab, which contains the list with the graphs that the user can draw, but it also serves as a store, where the states and values of the graphs are managed.

Inside the *index.js* file, we created the **registerGraph** function, that registers a graph component, with its data, options and settings, and constantly updates the *graphs* array in the *dashboard store*. The parameters of this function are:

Parameter title	Description
<i>name</i>	graph label, translatable string
<i>priority</i>	graph priority in the list of all graphs, lower means higher in the list of all graphs
<i>iconURL</i>	the relative path to the image representing the graph
<i>component</i>	the Vue component to display when the user chooses to draw a graph
<i>options</i>	additional options

Also here we create the functions **registerForSignal** and **emitSignal**, that will be used by the graphs and the connected device.

Here's a list of the graphs that are currently available in the application: *Gauge*, *Line*, *Speedometer*, *Thermometer*, *Vumeter*, *Switch*, *Slider*, *Extra*.

Each dashboard graph represents a new plugin, named “*dashboard.graph.name*”, where *name* represents the actual name of the graph.

The **views** folder contains 2 Vue components:

- *NameDialog.vue*, where we design the dialog opened when the user clicks on one graph from the list, allowing to customize the options and settings
- *NameGraph.vue*, where we use the **vue2-highcharts** module to draw a graph, according to the data entered by the user in the dialog; more details about the available Highcharts and the parameters required for each chart can be found [here](#).

The **index.js** file of each graph has the purpose to call the *registerGraph* function from the main plugin dashboard, where the *component* parameter is the *NameGraph* Vue component, and the *options* parameter is represented by an object where we define the *setup* property. Here, we call a function that opens the *NameDialog* component and updates each graph's setup options according to the data inputted by the user.

Of course, in the **package.json** file we have to specify that each dashboard.graph plugin consumes the main *dashboard* plugin.

6.4 Pin Layout plugin

The **Pin Layout** tab becomes visible for a user only when a board is connected to Wylodrin STUDIO, and it loads a “map” of the board and a legend of its pins. As we described in the *How to add a wyapp board* section, when we register this type of device, we call the *registerPinLayout* function.

registerPinLayout (*type*, *board*, *img*)

This function registers a customized pin layout image for the connected device. It's called each time you create a plugin for a new type of board. Depending on the type of the device or on the name of the board, the purpose of this function is to display the specified image within the Pin Layout tab.

Arguments

- **type** (*string*) – device type
- **board** (*string*) – board name
- **img** (*string*) – path to the pin layout image

For example, if you want to register a Raspberry Pi board, inside the corresponding plugin you will call this function:

```
studio.pin_layout.registerPinLayout ('wyapp', 'raspberrypi', 'plugins/devices/wyapp/  
↪devices/raspberrypi/data/img/pins-raspberrypi.png');
```

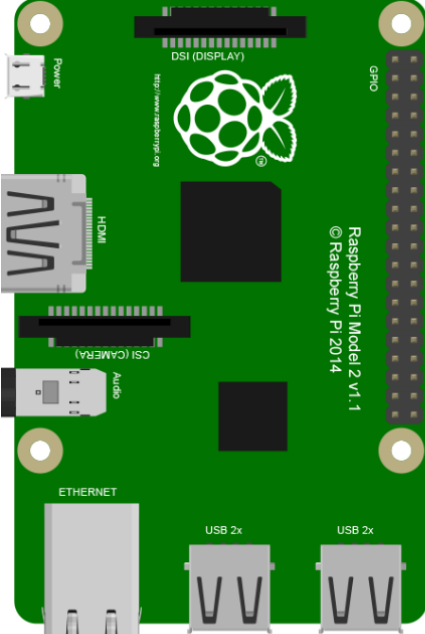
In this situation, the program will search for a device that has the 'wyapp' type, and the name of the corresponding board 'raspberrypi'.

However, you can register a pin layout only for a device type, and the selected image will be available for every device that has that type, no matter the name of the board:

```
studio.pin_layout.registerPinLayout ('wyapp', '', 'plugins/devices/wyapp/devices/
↳raspberrypi/data/img/pins-raspberrypi.png');
```

Once a Raspberry Pi board is connected to Wylidrin STUDIO, the Pin Layout tab will become available, and its content will be:

APPLICATION
DASHBOARD
NOTEBOOK
SCHEMATICS
PIN LAYOUT
SHELL



WiringPi	BCM(Name)	Physical	Physical	BCM(Name)	WiringPi
3v3 Power		1	2	5v Power	
8	BCM 2 (SDA)	3	4	5v Power	
9	BCM 3 (SCL)	5	6	Ground	
7	BCM 4 (GCLK0)	7	8	BCM 14 (TXD)	15
Ground		9	10	BCM 15 (RXD)	16
0	BCM 17	11	12	BCM 18 (PCM_C)	1
2	BCM 27 (PCM_D)	13	14	Ground	
3	BCM 22	15	16	BCM 23	4
3v3 Power		17	18	BCM 24	5
12	BCM 10 (MOSI)	19	20	Ground	
13	BCM 9 (MISO)	21	22	BCM 25	6
14	BCM 11 (SCLK)	23	24	BCM 8 (CE0)	10
Ground		25	26	BCM 7 (CE1)	11
BCM 0 (ID_SD)		27	28	BCM 1 (ID_SC)	
21	BCM 5	29	30	Ground	
22	BCM 6	31	32	BCM 12	26
23	BCM 13	33	34	Ground	
24	BCM 19 (MISO)	35	36	BCM 16	27
25	BCM 26	37	38	BCM 20 (MOSI)	28
Ground		39	40	BCM 21 (SCLK)	29

The Vue component of this plugin, *PinLayout.vue*, is designed to change the pin layout image dynamically, according to the device type and board, and to become enabled/disabled, depending on the status of the device (CONNECTED / DISCONNECTED).

6.5 Console and Shell plugins

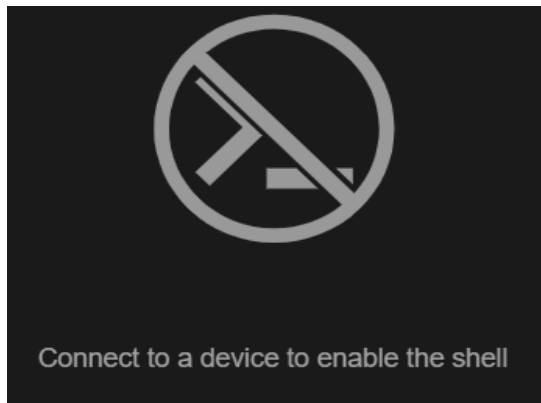
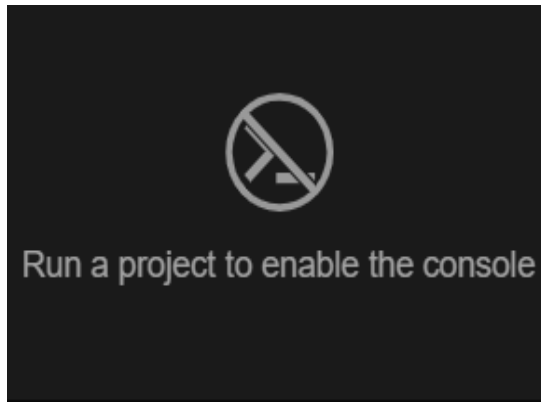
Both the Console and the Shell plugins depend on the *xterm* plugin.

The “xterm” plugin uses the **xterm** module in order to register a terminal that will allow the user to interact with a connected board.

The terminal has 2 implemented buttons:

- *clear*: clear the content of the terminal
- *reset*: reboot the terminal

Both functions belong to the **xterm Terminal**, that is initialized when a device is connected. If there is no connected device, the terminal won’t allow the user access and a replacement text will be displayed.



The Xterm Terminal functioning is based on events.

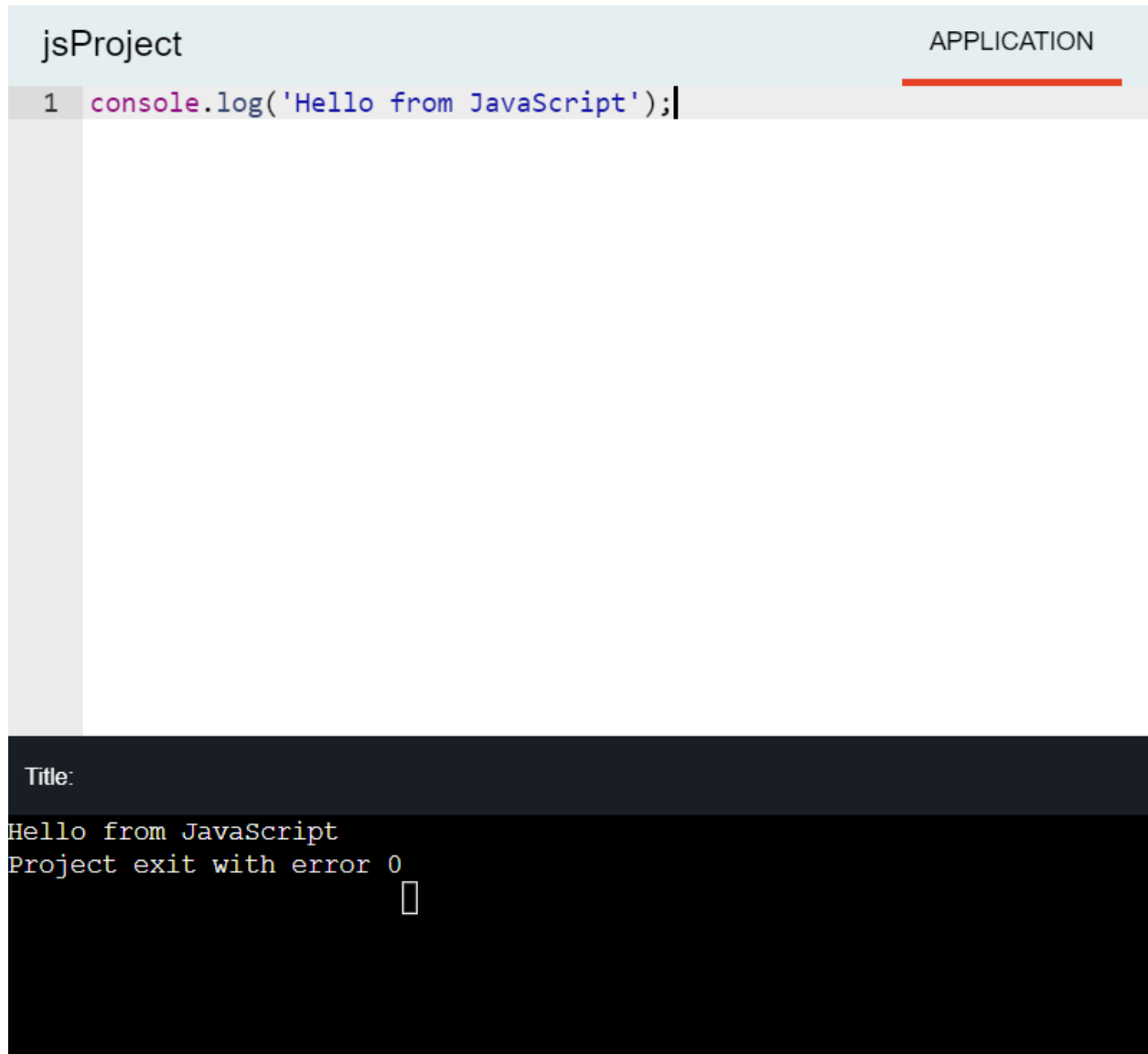
The **title** of the terminal is applied when a device is connected and it changes dynamically, according to the type of the board. For example, if a Raspberry Pi board is connected, the title will be detected and automatically set to the terminal as *pi@raspberrypi*.

When the user starts typing commands in the terminal, the **write** function is called in order to save all the inputted data into a buffer, unique for each terminal. We also retain the cursor position, to write the characters successively.

Another event is to **resize** the terminal and it has to be done at each update. The resizing action supposes to set the geometry of the terminal (number of columns and rows).

Both *console* and *shell* plugins have the functionalities of the described Terminal, so they have to consume the *xterm* plugin. However, there is a certain difference between the 2 components:

The purpose of the **Console** is to display a terminal that allows you to see the output of the projects that you run in the Application tab.



The **Shell** terminal represents the main component of the *Shell* tab, that allows you to send command directly to the board.

```
Title: pi@raspberrypi: ~
pi@raspberrypi:~ $ df -h -x squashfs
Filesystem      Size  Used Avail Use% Mounted on
/dev/root        3.6G  2.4G  1.1G   70% /
devtmpfs         460M    0  460M    0% /dev
tmpfs            464M    0  464M    0% /dev/shm
tmpfs            464M   6.3M  458M    2% /run
tmpfs            5.0M   4.0K   5.0M    1% /run/lock
tmpfs            464M    0  464M    0% /sys/fs/cgroup
/dev/mmcblk0p1   41M   23M   19M   55% /boot
tmpfs            93M    0   93M    0% /run/user/1000
pi@raspberrypi:~ $ ping www.google.com
PING www.google.com (172.217.18.164) 56(84) bytes of data.
64 bytes from fra15s29-in-f4.1e100.net (172.217.18.164): icmp_seq=1 ttl=52 time=29.4 ms
64 bytes from fra15s29-in-f4.1e100.net (172.217.18.164): icmp_seq=2 ttl=52 time=29.4 ms
64 bytes from fra15s29-in-f4.1e100.net (172.217.18.164): icmp_seq=3 ttl=52 time=29.4 ms
64 bytes from fra15s29-in-f4.1e100.net (172.217.18.164): icmp_seq=4 ttl=52 time=29.7 ms
```

6.6 Settings Plugin

The “*settings*” plugin consumes our *filesystem* plugin in order to save special files that contain various settings for our plugins. The filesystem is implemented differently for each version of the Wylidrin STUDIO application, but the main idea is to manage all the files and folders used inside the program.

In order to obtain the data that was written into a special settings file, we need to read the content of this file located inside a special settings folder.

The main functions of the settings plugin are:

storeSettings (*plugin*, *data*)

Save plugin settings.

For each plugin that this function is called for, we create an object with the data that will be stored and we use the filesystem function **writeFile** to save the parsed content into the *SETTINGS_FILE*

Arguments

- **plugin** (*string*) – plugin name
- **data** (*Object*) – plugin data

loadSettings (*plugin*)

Load plugin settings.

For the selected plugin, we display the data saved inside the special settings file.

Arguments

- **plugin** (*string*) – plugin name

Returns **Object** – - the data inside the settings file

loadValue (*plugin, name, value*)

Load value from settings.

We first load the settings from a chosen plugin using the **loadSettings** function. If the setting object exists and if there is a value for the chosen *name* property, we return that value.

Arguments

- **plugin** (*string*) – plugin name
- **name** (*string*) – property name
- **value** (*Object*) – the value to be associated to the property

Returns **string** – - the value in the settings file

storeValue (*plugin, name, value*)

Store value to settings.

The function first loads the existing settings of the selected project, then updates the chosen property of the object with the *value*.

Arguments

- **plugin** (*string*) – plugin name
- **name** (*string*) – property name
- **value** (*Object*) – the value to be associated to the property

How to write a plugin

7.1 Simple plugin

In this section, we will try to create a new plugin, called “**button.example**”, that will add a toolbar button which will show a notification when is clicked.

The purpose of this tutorial is to help you to better understand the idea of plugin, the steps that you need to follow, the structure and behavior of each component file, as they were explained in the [Architecture chapter](#).

The first step will be to create the *button.example* folder inside the *plugins* directory.

Each plugin contains 2 special folders:

The first one is the **data** folder, that has to be copied exactly as it is created in the *build* folder of the program. This **data** directory will include all the images used to represent the components of a plugin (tool buttons, icons), but also other additional files needed in order to make your plugin run properly.

The second special component is the **translations** folder, which will contain the translatable key strings from your plugin, and also their translations.

More details about how the translation function works can be found [here](#).

Only to exemplify the content of this folder, we'll create the **messages-en.json** (english language) and **messages-fr.json** (french language).

In our *index.js* file, you can notice that we used 2 strings having the following format: `'PLUGIN_STRING_TO_TRANSLATE'`, more precisely: `'EXAMPLE_BUTTON_NAME'` and `'EXAMPLE_BUTTON_NOTIFICATION_TEXT'`. It means that this key-strings have to be included in both our translation files.

As you can see in the [Translations](#) chapter, the value that the key string will receive has to be an object with 2 properties: *message* (the actual translation), *description* (a short definition of the string to translate).

By the end, your **messages-ln.json** (ln = language) files should look like this:

“messages-en.json”:

```
{
  "EXAMPLE_BUTTON_NAME": {
    "message": "Notify",
    "description": "This button pops-up a notification."
  },
  "EXAMPLE_BUTTON_NOTIFICATION_TEXT": {
    "messages": "You have successfully created your button!",
    "description": "This is the notification text when the user clicks_
↪the button."
  }
}
```

“messages-fr.json”:

```
{
  "EXAMPLE_BUTTON_NAME": {
    "message": "Notifier",
    "description": "This button pops-up a notification."
  },
  "EXAMPLE_BUTTON_NOTIFICATION_TEXT": {
    "messages": "Vous avez créé le bouton avec succès",
    "description": "This is the notification text when the user clicks_
↪the button."
  }
}
```

Then, we’ll add the **package.json** file. As mentioned before, the content of this type of file has to be an object with the following properties:

Property title	Description	Required / Optional	Default value
<i>name</i>	the name of the plugin (“button.example”)	required	-
<i>version</i>	0.0.1	required	“0.0.1”
<i>main</i>	the main file of the plugin, that will be “index.js”	required	“index.js”
<i>plugin</i>	an object where we specify the characteristics of the plugin	required	-

The properties of the “*plugin*” component are:

Property title	Description	Required / Optional	Default value
<i>consumes</i>	we specify from which other plugins our plugin uses exported functions (required “ <i>workspace</i> ”)	required	[“workspace”]
<i>provides</i>	we specify if our plugin functions will be exported (“ <i>example_button</i> ”)	optional	[]
<i>target</i>	for which version of the program the plugin should be working: browser or electron	required	-

Finally, the content of our package.json will be:

```
{
  "name": "button.example",
  "version": "0.0.1",
  "main": "index.js",
  "private": false,
  "plugin": {
    "consumes": ["workspace"],
    "provides": ["button_example"],
    "target" : ["browser", "electron"]
  }
}
```

The next step is to create the main file, called **index.js**.

If you already read [this section](#), you probably noticed that in the **index.js** file we should’ve imported first the **.vue** files from the **views** folder. In this plugin tutorial, we only register a simple button, which means that we don’t need a **.vue** file to design a specific Vue component, so the **views** folder will also be missing.

Therefore, we’ll only need to initiate a **studio** variable to *null* and to create an empty object called **button example**.

After that, we have to export a *setup* function, its parameters being:

Property title	Description	Required / Optional	Default value
<i>options</i>	additional options	optional	null
<i>imports</i>	all the functions that our plugin collects from the plugins that it consumes (in our case, the functions exported by <i>workspace</i>)	required	-
<i>register</i>	a function that will register the plugin object	required	-

Inside this function, the **studio** variable instantiated before will receive the **imports** value.

After that, we need to register our button, so we’ll call the workspace function **registerToolBarButton**, which will have the following parameters:

<code>‘BUTTON_EXAMPLE_NAME’</code>	the name of our button, a key string that will be translated
<code>20</code>	integer number representing the priority of our button in the list of all toolbar buttons
<code>() => studio.workspace.showNotification</code>	the action that will be performed when the user clicks on this button
<code>‘plugins/button.example/data/img/button.png’</code>	the relative path to the image that will represent our button

The **showNotification** function is also called from the workspace and its parameters are:

<code>'BUTTON_EXAMPLE_NOTIFICATION_TEXT'</code>	the key that will be translated and will represent the text of our notification
<code>'success'</code>	the notification type

By the end, our **index.js** file should look like this:

```
let studio = null;
let button_example = {};

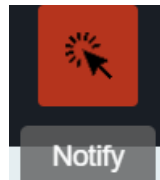
export function setup(options, imports, register)
{
    /* Collect the objects exported by the consumed plugins */
    studio = imports;

    /* Create a toolbar button that will display a notification */
    studio.workspace.registerToolBarButton ('EXAMPLE_BUTTON_NAME', 20,
        () => studio.workspace.showNotification ('EXAMPLE_BUTTON_NOTIFICATION_
        ↪TEXT'),
        'plugins/button.example/data/img/button.png');

    /* Register the object that this plugin will provide */
    register(null, {
        button_example: button_example;
    })
}
```

As you noticed above, when we registered the image corresponding to our button, we specified its relative path, which includes some additional folders in our *button.example* plugin.

To test if you successfully created your first plugin, you have to rebuild the program using the 2 commands for electron **npmx webpack**, then **npm start**.




 A blue rectangular banner with a white information icon on the left, the text "You have successfully created your button!" in the center, and a white close icon on the right.

If you want to test this plugin, you will have to search for **“button.example”** in the *docs/examples* folder and copy it inside the *source/plugins* folder, then rebuild the application to make the new plugin available.

7.2 How to create a device plugin

This type of plugin allows you to add and use a new device to the Wylodrin STUDIO platform, so you need to properly register its functions and characteristics.

Let’s suppose that you want to create your own device plugin, called **“device.awesome”**.

The **data** folder should contain all the images that you need to represent the device (the icon displayed in the list of available devices) and its features (for example, the DeviceToolButtons), but also, if needed, the additional files that you’ll use to make your device run projects.

The **views** folder has to include every Vue component relied to your device, for example: disconnect, device settings or device manager dialogs.

For this example, we will create the **AwesomeDisconnectDialog.vue** component, that will contain the button that disconnects the device:

```
<template>
  <v-card class="disconnect">
    <v-tooltip>
      <template #activator="data">
        <v-btn @click.stop="disconnect" class="icon-btn" ref=
↵ "reference">
          
            </v-btn>
          </template>
          <span>{{ $t('DEVICE_AWESOME_DISCONNECT') }}</span>
        </v-tooltip>
      </v-card>
    </template>

<script>
  /* The actual code goes here */
</script>
```

The *script* part will define the *disconnect* function and also an *esc* function, that will close the dialog containing the Disconnect Button when the user presses the ‘Esc’ key:

```
export default {
  name: 'AwesomeDisconnectDialog',
  methods: {
    disconnect ()
    {
      /* Send the 'disconnect' tag */
      this.$root.$emit ('submit', {
        disconnect: 'disconnect'
      });
    },
    esc ()
    {
      /* Emit the 'submit' signal from the child component to
      ↪ notify the parent that the dialog has to be closed */
      this.$root.$emit ('submit');
    }
  }
}
```

The **package.json** file will have the classic format, but if it’s necessary the “plugin” object will require an additional property, called “**optional**”, where you will specify if the plugin consumes the *console* or the *mqtt* plugins.

For the example created, it won’t be necessary, so the content of this file will be:

```
{
  "name": "device.awesome",
  "version": "0.0.1",
  "main": "index.js",
  "private": true,
  "plugin": {
    "consumes": ["workspace", "projects"],
    "provides": [],
    "target": ["electron"]
  }
}
```

The **translations** folder will also have the usual structure, including the *messages-ln.json* files with the unique keys that you used in your device plugin, for each language of the program.

```
{
  "DEVICE_AWESOME_DISCONNECT": {
    "message": "Disconnect",
    "description": "This button is used to disconnect a device."
  }
}
```

The main file **index.js** is the most important for this type of plugin, as its purpose is to include all the functions and characteristics that will make your device work.

You have to begin with importing all the Vue components that you created, and also all the modules and packages that your device requires in order to work properly.

For the “device_awesome” plugin, the header of this file could look like this:

```
/* Here you will import all the modules required for the functioning of your device */

import AwesomeDisconnectDialog from './views/AwesomeDisconnectDialog.vue';

import { EventEmitter } from 'events';
import { connect } from 'http2';

let deviceEvents = new EventEmitter ();

let awesome_module = null;

let studio = null;
let workspace = null;
let devices = [];

let awesomeDevices = [];

let connections = {};
```

After that, you will create the functions needed to search and update your device type:

loadDevice: uses a specialized module to scan the operating system of the client and search for your type of device.

```
function loadAwesome ()
{
    try
    {
        /* Any module that will allow you to find the type of device you have_
        ↪chosen */

        return require ('awesome_module');
    }
    catch (e)
    {
        studio.workspace.error ('device_awesome: Awesome is not available '+e.
        ↪message);

        return {
            list: function ()
            {
                return [
                ];
            }
        };
    }
}
```

listDevice: will try to return a list of the available devices, if they can be found.

```

async function listAwesome ()
{
    let ports = [];
    try
    {
        ports = await awesome_module.list ();
    }
    catch (e)
    {
        studio.workspace.error ('device_awesome: failed to list awesome '+e.
↪message);
    }
    return ports;
}

```

updateDevices: simply call the workspace *updateDevices* function.

```

function updateDevices()
{
    workspace.updateDevices ([...devices, ...awesomeDevices]);
}

```

searchDevices: checks systematically the list with all the available devices found, trying to find those having the name or the description fitting your type of device, then adds a new object to the *devices* array, with the relevant properties: unique *id*, *name*, *description*, *address*, *priority*, *icon*, type of *board*, type of *connection*, and others additional options.

```

function search ()
{
    if(!discoverAwesomeDevicesTimer)
    {
        discoverAwesomeDevicesTimer = setInterval (async () => {
            let awesome_devices = await listAwesome ();
            devices = [];
            for(let awesomeDevice of awesome_devices)
            {
                ↪/* Search only for the devices that have the same_
specifications as your Awesome Device, array and set its properties.*/

                devices.push(awesomeDevice);
            }
            updateDevices ();
        }, 5000);
    }
}

```

Inside the *setup* function, you first have to obtain the list of devices that fit your *awesome* type:

```

export function setup (options, imports, register)
{
    studio = imports;
    awesome_module = loadAwesome();
    search();

    ↪/* Code explained below */
}

```

After that, you will create the object you will register and export for your plugin, its properties being the functions that will help the user manage your device on the Wylidrin Studio platform:

defaultIcon: correlates a default icon to a device that doesn't have any particular image already attached

```
defaultIcon ()
{
    return 'plugins/device.awesome/data/img/icons/awesome.png';
}
```

registerForUpdate: registers to receive updates for a device

```
registerForUpdate (device, fn)
{
    deviceEvents.on ('update:'+device.id, fn);
    return () => deviceEvents.removeListener ('update:'+device.id, fn);
}
```

getConnections: returns the connections array for every unique device id

```
getConnections ()
{
    let connections = [];
    for (let deviceId in connections)
    {
        connections.push (connections[deviceId].device);
    }
    return connections;
}
```

connect: connects the device to Wylodrin Studio; if there is no connection previously created for the current unique id of the device, you should create a data transport path conforming with the type of your device;

```
connect (device, options)
{
    /* Here goes the actual code that you will write in order to connect the
    ↪device. */

    setTimeout (() => {
        device.status = 'CONNECTED';
    }, 1000);
}
```

after that, according to the current status, you will bring up to date your device, using the *updateDevices* function and you will set up its functioning characteristics.

The device statuses are:

DISCONNECTED	the device is offline
CONNECTING	trying to connect
SYNCHRONIZING	trying to synchronize with the device
CONNECTED	the device is online
ISSUE	there is some issue, the system is partially functional
ERROR	there is an error with the system

disconnect: opens a dialog where the user chooses the way he wants to disconnect the device; the methods of disconnection are:

- *StandBy* -
- *Disconnect* -

- *Turn-Off* -

```
disconnect(device, options)
{
    /* Here goes the actual code that you will write in order to connect the_
    ↪device. */
    setTimeout(() => {
        device.status = 'DISCONNECTED';
    }, 1000);
}
```

After creating the new device object, you have to register it using the workspace function *registerDeviceDriver*.

```
workspace = studio.workspace.registerDeviceDriver('awesome', device_awesome);
```

Here you can also generate the specific buttons for your type of device, using also an workspace function: *registerDeviceToolButton*.

For the *awesome device* we create a **Run** button, that will run the code written by the user in the current project.

```
workspace.registerDeviceToolButton('DEVICE_AWESOME_RUN', 10 async () => {
    let device = studio.workspace.getDevice ();

    /* Here goes the actual code that will make your device run a project */
    console.log('Run');
    }, 'plugins/device.awesome/data/img/icons/run-icon.svg',

    /* The additional options that make the Run Button visible and enabled only if_
    ↪there is a connected device
    and its type is "awesome" */
    {
        visible () {
            let device = studio.workspace.getDevice ();
            return (device.status === 'CONNECTED' && device.connection_
            ↪=== 'awesome');
        },
        enabled () {
            let device = studio.workspace.getDevice ();
            return (device.status === 'CONNECTED' && device.connection_
            ↪=== 'awesome');
        },
        type: 'run'
    });
```

Also, if your device interacts with the *console* or the *mqtt* server, you will have to create some specific functions that will establish the data transfer protocol.

At the end of the setup function, we register the *device_awesome* object:

```
register(null, {
    device_awesome
});
```

If you want to test this plugin, you will have to search for “**device.awesome**” in the *docs/examples* folder and copy it

inside the *source/plugins* folder, then rebuild the application to make the new plugin available.

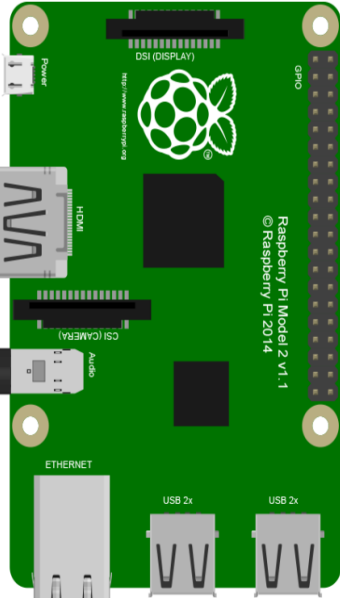
7.3 How to add a wyapp board

If you're trying to add a new board plugin, our “*device.wyapp.raspberrypi*”, “*device.wyapp.beagleboneblack*” and “*device.wyapp.udooneo*” plugins may serve as a support for you.

In the **index.js** file, inside the *setup* function, you need to create an event, so when the board is ‘*ready*’, you call the **registerPinLayout** function from our “*pinlayout*” plugin. The purpose of this function is to register the pins of your board in the **Pin Layout** tab, using the appropriate images that you saved in the *data* folder of our plugin.

For example, if we are connected to a Raspberry Pi, the content of the Pin Layout tab will be:

APPLICATION DASHBOARD NOTEBOOK SCHEMATICS **PIN LAYOUT** SHELL



WiringPi	BCM(Name)	Physical	Physical	BCM(Name)	WiringPi
3v3 Power		1	1	5v Power	
8	BCM 2 (SDA)	3	3	5v Power	
9	BCM 3 (SCL)	5	5	Ground	
7	BCM 4 (GCLK0)	7	7	BCM 14 (TXD)	15
Ground		9	9	BCM 15 (RXD)	16
0	BCM 17	11	11	BCM 18 (PCM_C)	1
2	BCM 27 (PCM_D)	13	13	Ground	
3	BCM 22	15	15	BCM 23	4
3v3 Power		17	17	BCM 24	5
12	BCM 10 (MOSI)	19	19	Ground	
13	BCM 9 (MISO)	21	21	BCM 25	6
14	BCM 11 (SCLK)	23	23	BCM 8 (CE0)	10
Ground		25	25	BCM 7 (CE1)	11
BCM 0 (ID_SD)		27	27	BCM 1 (ID_SC)	
21	BCM 5	29	29	Ground	
22	BCM 6	31	31	BCM 12	26
23	BCM 13	33	33	Ground	
24	BCM 19 (MISO)	35	35	BCM 16	27
25	BCM 26	37	37	BCM 20 (MOSI)	28
Ground		39	39	BCM 21 (SCLK)	29

The next step is to create an object having your new board name, with the next functions:

iconURL() => the image corresponding to your board

found(device) => if a device was found, you can modify some of its properties

update(device) => update a device, modify some of its properties

run(project) => modify the project before run

The final step is to register your board and, if it's necessary, the blocks that you'll use, from the “*editor_visual*” plugin. For example, if you want to register a *raspberrypi* board, you should use this function:

```
registerBoard ('raspberrypi', raspberrypi);
```

7.4 How to write an editor plugin

The purpose of an editor plugin is to create a code editor, correlated to our “*projects*” plugin. The editor will allow the user to open different type of files created or imported within the tree structure of a project.

If you are creating an editor plugin, we recommend you to name the folder “**projects.editor**”, followed by the name of your editor.

For example, in this tutorial we will create an *awesome* editor, that will display the content of the files having the *.aws* extension.

The first step is to create the **package.json** file, which will have the classic structure:

```
{
  "name": "projects.editor.awesome",
  "version": "0.0.1",
  "main": "index.js",
  "private": true,
  "plugin": {
    "consumes": ["workspace", "projects"],
    "provides": [],
    "target": ["electron", "browser"]
  }
}
```

In this example, the editor will use functions only from the “workspace” and “projects” plugins, but you are free to “consume” any other plugin required by your editor.

After that, we will add the **views** folder, where you will design the Vue components for your editor, in this example **AwesomeEditor.vue**. In the *template* section, you will actually add the tags required by the code editor, while in the *script* part you will handle the functions that your editor will perform in order to display the content of the supported files.

```
<template>
  <!-- Here goes the design of your editor -->
</template>
```

```
/* <script> */

import path from 'path';

export default {
  name: 'AwesomeEditor',

  /* We pass the 'project' (path to the current project) and 'filename' (name_
  ↳ of the opened file, including extension)
  ↳ in order to read the content of the file and handle it depending on the_
  ↳ type of extension (continues on next page)
```


(continued from previous page)

```

    */
    props: ['project', 'filename'],
    data() {
      return {
        /* All the variables you will use in the template section */
      },
    },
    methods: {
      /* Code of all the function you will use in the template section */
    },
    watch: {
      filename: {
        immediate: true,
        async handler() {
          /* Full path to the current file */
          let filePath = path.join(this.project.folder, this.
↪filename);

          /*Extension of the current file */
          let extension = this.filename.substring(this.filename.
↪lastIndexOf('.')).substring(1);

          /* Get the content of the current file */

          let content = await this.studio.filesystem.
↪readFile(filePath);

          /* Here goes the code for your file editor */

        }
      }
    }
  }
}
/* </script> */

```

The final step is to create the **index.js** file, where you will register your editor. The structure of this file should look like this:

```

import AwesomeEditor from './views/AwesomeEditor.vue';

export default function setup (options, imports, register)
{
  const studio = imports;
  studio.projects.registerEditor('EDITOR_AWESOME', ['aws'], AwesomeEditor);

  register (null, {});
}

```

The *AwesomeEditor* is registered using the **registerEditor** function:

registerEditor (*name*, *languages*, *component*, *options*)

This function registers a new type of editor.

The editor has a *name*, which is a translatable string that will be displayed as the title of the editor, *languages*,

which represent the array with all the supported programming languages id's or file extensions, and a Vue *component*, representing the actual content and design of the editor tab.

Arguments

- **name** (*string*) – the name/id of the editor
- **languages** (*Array.<string>*) – the editor languages
- **component** (*Vue*) – the component to display
- **options** (*array*) – the editor options

Returns **boolean** – - true if successful, false otherwise

Examples:

```
registerEditor('EDITOR_ACE', ['py', 'js'], Ace);
```

7.5 How to write a language plugin

The purpose of this type of plugins is to register a new programming language that will be supported by the Wylidrin Studio IDE.

For example, we'll try to add a new programming language, called “MyAwesomeLanguage”, with the “.aws” extension:

As you can notice, the name of this type of plugins should begin with “*language.*”, which will be followed by the actual name of the programming language that you want to register, which means that you will have to create a new folder, “**language.awesome**”.

As any other plugin, it's required to have a *package.json* file, having the classic format. It's necessary to mention that this type of plugin **consumes** both “*workspace*” and “*projects*” plugins, and their **target** are both “*electron*” and “*browser*”.

So, the content of your package.json should look like that:

```
{
  "name": "language.awesome",
  "version": "0.0.1",
  "main": "index.js",
  "private": true,
  "plugin": {
    "consumes": ["workspace", "projects"],
    "provides": [],
    "target": ["electron", "browser"]
  }
}
```

The language plugin doesn't have any Vue component, so we don't have to create the **views** folder, but we need the **data** folder to save a characteristic image for the programming language. Let's pick as example for our *language.awesome* plugin, an icon that we will save in the **data/img** folder:



Inside the main file, **index.js**, we obviously need to initialize the *studio* variable to null, and inside the *setup* function it will receive all the imported functions from the “workspace” and “projects” plugin.

The next step is to create the **awesome** object, containing the options of our programming language:

```
let studio = null;

export default function setup (options, imports, register)
{
    studio = imports;

    let awesome = {

        /* Create the main file of each project, "main.aws" */
        async createProject(name) {
            await studio.projects.newFile(name, '/main.aws', 'print ("Hello_
↳from Awesome") ');
        },

        /* Return the name of the default file */
        getDefaultFileName() {
            return '/main.aws';
        },

        /* Return the name of the default run file */
        getDefaultRunFileName() {
            return '/main.aws';
        },

        /* Return the content of the makefile */
        getMakefile(project, filename) {
            if (filename[0] === '/')
                filename = filename.substring (1);

            return 'run:\n\tawesome main.aws';
        },
    };
}
```

The next step is to register the new programming language, using the function *registerLanguage*:

```
studio.projects.registerLanguage('awesome', 'awesome', 'plugins/language.awesome/data/
↳img/project.png', 'plugins/language.awesome/data/img/awesome.png', awesome);
```

where the last parameter represents the *awesome* object we created before.

If you want to test this plugin, you will have to search for “**language.awesome**” in the *docs/examples* folder and copy it inside the *source/plugins* folder, then rebuild the application to make the new plugin available.

7.6 How to add a language addon plugin

This type of plugin modifies the language plugin for certain devices. For instant, we are using it for visual and rpk. To design your own language addon, you will have to create a new plugin folder, called “*language.visual.*”, followed by the type of the device you want the language addon for.

For example, let’s say that you want to create an addon for your *Awesome* device and you need to create a new plugin, called **language.visual.awesome**

The first step is to create a new folder, **visual**, where you will add *.js* files.

You will also have to create a *toolbox.xml* file, where you will include the actual design of the blocks you want to be available for your device.

The **index.js** file will first import the *xml* module and the *toolbox.xml* file, the second one as a string, using the *raw-loader* module. More details about this webpack loader can be found [here](#).

```
import xml from 'xml-js';
import toolboxStr from 'raw-loader!./visual/toolbox.xml';
```

Then, you will import the code and the blocks from the *.js* files included in the *visual* folder.

```
let blocks = require ('./visual/definitions_for_awesome.js');
let code = require ('./visual/code_for_awesome.js');
```

The *setup* function will register the changes you made for your device, using the projects function *registerLanguageAddon*.

```
let studio = null;
export function setup (options, imports, register)
{
    studio = imports;

    studio.projects.registerLanguageAddon ('visual', 'awesome', 'awesome', {
        getDefaultRunFileName ()
        {
            return '/main.visual.js';
        },

        sourceLanguage ()
        {
            return 'awesomelanguage';
        }
    });
};
```

(continues on next page)

(continued from previous page)

```

    let toolbox = xml.xml2js (toolboxStr);
    studio.editor_visual.registerBlocksDefinitions ('awesome', blocks, code,
↪toolbox, {type: 'awesome', board: 'awesome'});

    register (null, {});
}

```

As you can notice, the final step is to parse the toolbox string imported before and then to register the blocks using the **registerBlocksDefinitions** function from the *projects.editor.visual* plugin.

The parameters of this function are:

Property title	Description	Required / Optional	Default value
<i>id</i>	the id of the device	required	-
<i>blocks</i>	the blockly visual blocks	required	-
<i>code</i>	the blockly code	required	-
<i>toolbox</i>	the parsed toolbox string	required	-
<i>options</i>	additional options, an object where you can specify the device type and the board	optional	{}

Of course, you also need to have a **package.json** file, where you should mention that your language addon plugin also consumes “editor_visual”, because it’s using the *registerBlockDefinitions* function.

```

{
  "name": "language.visual.awesome",
  "version": "0.0.1",
  "main": "index.js",
  "private": true,
  "plugin": {
    "consumes": ["workspace", "projects", "editor_visual"],
    "provides": [],
    "target": ["electron"]
  }
}

```

If you want to test this plugin, you will have to search for “**language.visual.awesome**” in the *docs/examples* folder and copy it inside the *source/plugins* folder, then rebuild the application to make the new plugin available.

CHAPTER 8

Translations

Each plugin has a **translations** folder, where you can find the **messages-ln.json** files, one for each language available in our application. These files contain an object with a list of key-value sets.

In the *.vue* files you will use strings on different purposes (for example, to name a button) and you will need to update their translation according to the language you choose in the app. This action is possible using our translation function **\$t**, that can be used in 2 forms:

1. Vue template

```
{{ $t('PLUGIN_STRING_TO_TRANSLATE') }}
```

where *PLUGIN* will be the name of your plugin and *STRING_TO_TRANSLATE* is a keyword for the actual text that you want to add.

2. Code

```
this.$t(text)
```

where *text* is a parameter of a function (for example *showNotification*) that includes the translation function.

We use **this.\$t(text)** so the program knows to translate the parameter *text*, regardless of the value it receives. When we call the *showNotification* function, *text* will also receive a keyword, for example:

```
showNotification('PLUGIN_STRING_TO_TRANSLATE');
```

In both situations, 'PLUGIN_STRING_TO_TRANSLATE' is a key that you will include in the *messages-ln.json* file, for each language. Its corresponding value is a new object, that contains a **message** (the translation itself) and a **description**.

For example, let's say that in your `message-en.json` (English language) you want to translate the word 'Close', that will be attached to a button.

```
{
  "MYNEWPLUGIN_CLOSE": {
    "message": "Close",
    "description": "This button is used to close the current window."
  }
}
```

As you can imagine, in your `messages-fr.json` (French language), you'll have:

```
{
  "MYNEWPLUGIN_CLOSE": {
    "message": "Fermer",
    "description": "This button is used to close the current window."
  }
}
```

8.1 Load and Send translation files

Inside the Wyliodrin Studio repository, you will find a directory named **tools**, which includes a **translation** sub-directory, with a **translation.js** main file. Here, you have 2 options to run this file:

```
node translation.js
```

This command joins all the key-value sets from all the existing plugins, for each language, into the `messages-ln.json` files from the current **translation** folder. It also checks for errors through all these files, using as reference file the english translation, and it lets you know if there are missing or duplicate keywords in a certain language.


```
node translation.js send
```

Compiling the code with the **‘send’** argument helps you split all the translations in a *messages-ln.json* file according to the plugin related to each key-value set. It also copies the description from the english translation and applies it to the corresponding keyword for every other language.

Dialogs and Notifications

In the “*workspace*” plugin you will find, additionally to the functions presented in the API sections, some functions designed to create and display some customized pop-ups, like dialogs, prompts and notifications.

9.1 Dialogs

A dialog is a component that informs users about a specific task and may contain important informations, require decisions, or involve multiple actions or inputs. It can usually be used to collect data from the user.

showDialog (*title*, *component*, *options*, *buttons*, *values*={})

This function shows a dialog that can contain informations about an application component or that can require user actions.

The dialog will have a translatable *title*, displayed on the top of the box, a Vue *component* specifically designed to collect the required data from the user, additional *options* and *buttons* to customize the dialog window, and the *values* option that allow the translation of some system variables the user is working with.

Arguments

- **title** (*string/object*) – the title of the dialog window
- **component** (*Vue*) – the Vue component to display
- **options** (*Object*) – additional like width
- **buttons** (*Array.<Object>*) – the array of buttons to display
- **values** (*Object*) – values to insert into the translatable text

For example, having the *Simple plugin* created, let’s say that when the button is clicked, you want to open a simple dialog with an input text area and a “Close” button. The content of the *ButtonDialog.vue* component will be:

```
<template>
  <v-card>
    <v-card-text>
      {{$t('BUTTON_EXAMPLE_INPUT_TEXT')}}
      <v-text-field></v-text-field>
    </v-card-text>

    <v-card-actions>
      <v-btn text @click="close">Close</v-btn>
    </v-card-actions>
  </v-card>
</template>
```

Inside the **script** section, you will define the methods that your component needs:

```
export default {
  name: 'ButtonDialog',
  data() {
    return {

    },
  },
  methods: {
    close() {
      this.$root.$emit ('submit', undefined);
    }
  }
}
```

The *index.js* file will have the following structure:

```
import ButtonDialog from './views/ButtonDialog.vue';

let studio = null;

export function setup(options, imports, register)
{
  studio = imports;

  /* Register a toolbar button that on click will reveal a dialog with the
  ↪specified title, image and component */
  studio.workspace.registerToolbarButton ('BUTTON_EXAMPLE_NAME', 20,
    () => studio.workspace.showDialog ('BUTTON_EXAMPLE_DIALOG_TITLE',
  ↪ButtonDialog),
    'plugins/button.example/data/img/button.png');

  register(null, {
    button_example: button_example
  });
}
```

The *title* parameter is not mandatory when you call the **showDialog** function, because you can choose the title of a dialog box within the Vue file that designs this component.

For example:

```
<template>
  <v-card>
```

(continues on next page)

(continued from previous page)

```

        <v-card-title>
            {{ $t('BUTTON_EXAMPLE_DIALOG_TITLE') }}
        </v-card-title>

        <v-card-text>
            {{ $t('BUTTON_EXAMPLE_INPUT_TEXT') }}
            <v-text-field></v-text-field>
        </v-card-text>

        <v-card-actions>
            <v-btn text @click="close">Close</v-btn>
        </v-card-actions>
    </v-card>
</template>

```

The **script** section will have the same structure as before, while within the **index.js** file you will have to register your button as it follows:

```

studio.workspace.registerToolbarButton ('BUTTON_EXAMPLE_NAME', 20,
    () => studio.workspace.showDialog (ButtonDialog),
    'plugins/button.example/data/img/button.png');

```

As you can notice, the **showDialog** function will use only the *ButtonDialog* component as parameter.

In both situations the result will be the same:

Button Dialog Title

Please input your text here:

CLOSE

9.2 Prompts

A prompt is actually a dialog box that requires a user decision. A prompt box is often used if you want the user to input a value before entering a page, for example write a text or click on a button that will perform a certain action.

showPrompt (*title*, *question*, *original*, *action*, *value*={})

This function shows a customized prompt that waits for user input and collects data.

A prompt has a *title*, that is located at the top of the box and it indicates the purpose of the prompt, a *question*, representing the requirement addressed to users, an *original* value contained in the input area, an *action* to be performed, and the *values* option that allow the translation of some system variables the user is working with.

Arguments

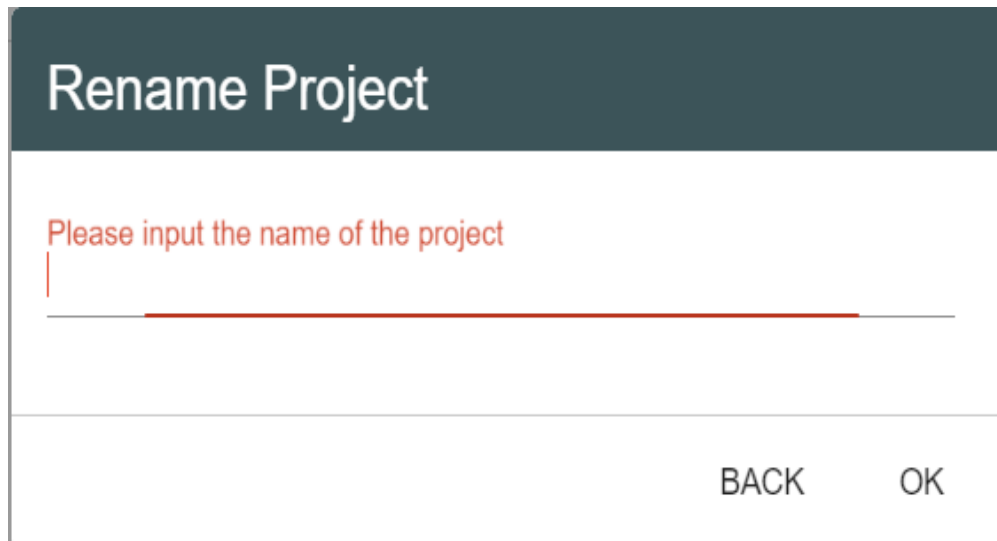
- **title** (*string*) – the translatable title of the prompt to be displayed
- **question** (*string*) – the translatable question of the prompt to be displayed
- **original** (*string*) – the original translatable content of the input area
- **action** (*Object*) – the action performed
- **value** (*Object*) – values to insert into the translatable text

Examples:

```
showPrompt ('PROJECT_RENAME_PROJECT', 'PROJECT_NAME_PROMPT', '');
```

This prompt is used to rename a project. The 'PROJECT_RENAME_PROJECT' is a translatable key string that corresponds to the title of the prompt (*Rename Project*) and 'PROJECT_NAME_PROMPT' represents the question or the statement addressed to the user (*Please input the name of the project*). Both key strings have to be included within the translations files.

The **showPrompt** function will return the value inputted by the user if he will click on *OK* and null otherwise, so that you can perform different actions depending on its answer.



showConfirmationPrompt (*title*, *question*, *values*={})

This function shows a customized prompt containing a simple question and waiting for a *Yes/No* response.

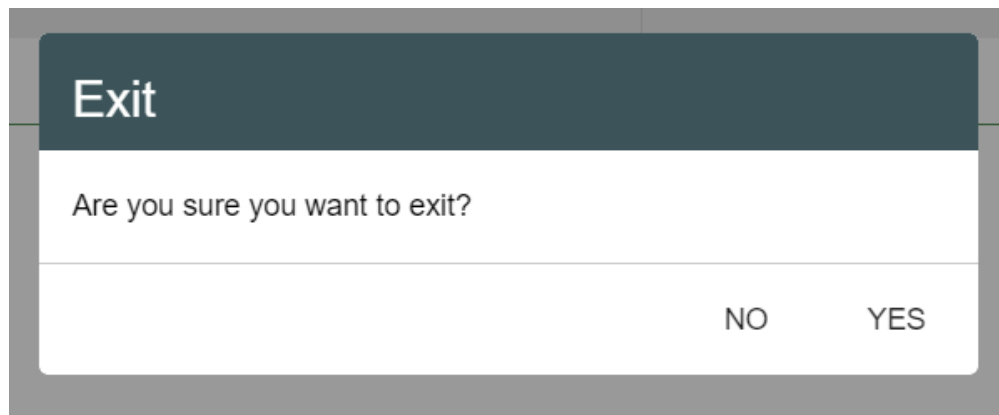
This prompt also has a *title*, that is located at the top of the box and it indicates the purpose of the prompt, a *question*, addressed to users in order to confirm an action that will be performed, and the *values* option that allow the translation of some system variables the user is working with.

Arguments

- **title** (*string*) – the translatable title of the prompt to be displayed
- **question** (*string*) – the translatable question of the prompt to be displayed
- **values** (*Object*) – values to insert into the translatable text

Examples:

```
showConfirmationPrompt('EXIT', 'WORKSPACE_TOOLBAR_EXIT_QUESTION');
```



9.3 Notifications

The notifications are simple pop-ups that inform the user about unauthorized actions, required operations or system processes.

The possible types for a notification are: *info*, *success*, and *warning*, and each type has a distinct color.

showNotification (*text*, *values*={}, *type*, *timeout*=6000)

This function shows a notification that will inform the user about the current application state.

The notification will have a *text* content, that will be translated according to the current language of the program, but it can also contain the name of one system variable the user is working with. This variable is included in the *values* object in order to be translated, because its value can change dynamically. Each notification also has a *type*, that will update the color of the notification box, and a *timeout* to be visible for the user, its default value being 6 seconds.

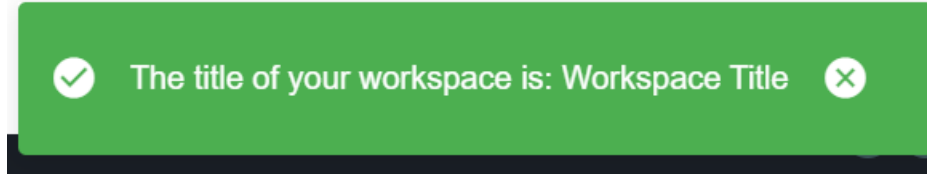
Arguments

- **text** (*string*) – the translatable ID of the text to be displayed

- **values** (*Object*) – values to insert into the translatable text
- **type** (*string*) – the notification type: info/success/warning
- **timeout** (*number*) – timeout until the notification is dismissed automatically (0 for never)

Examples:

```
studio.workspace.showNotification ('TRANSLATED_TEXT_ID', {title: 'the title'},
↪ 'success', 5000);
```



In this situation, “title” is a variable that represents the title of the notification and will be included in the *messages-
In.json* translation files as it follows:

```
{
  "TRANSLATED_TEXT_ID": {
    "message": "The title of your workspace is: {title}",
    "description": "Text of the notification the user created."
  }
}
```

title will be the actual name of your workspace, in this example: *Workspace Title*.

showError (*text*, *value*={}, *timeout*=6000)

This function sends an error notification in the application, when the user is trying to perform an unauthorized action.

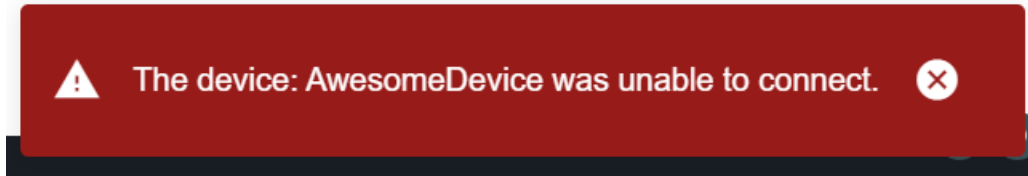
The error notification will have a *text* content, that will be translated according to the current language of the program, but it can also contain the name of one system variable the user is working with. This variable is included in the *values* object in order to be translated, because its value can change dynamically, and a timeout to be visible for the user, its default value being 6 seconds.. In opposition to a basic notification, the default *type* is *error*.

Arguments

- **text** (*string*) – the translatable ID of the text to be displayed
- **value** (*Object*) – values to insert into the translatable text
- **timeout** (*number*) – timeout until the notification is dismissed automatically (0 for never)

Examples:

```
studio.workspace.showError ('TRANSLATED_TEXT_ID', {title: 'the title'}, 5000);
```

Similar to *showNotification*, “title” is a variable that represents the title of the error notification and will be included in the *messages-ln.json* translation files as it follows:

```
{
  "TRANSLATED_TEXT_ID": {
    "message": "The device: {title} was unable to connect.",
    "description": "Text of the notification the user created."
  }
}
```

title will be the name of the device you are trying to connect, in this example: *AwesomeDevice*.

10.1 QEMU Based

10.1.1 Install QEMU

The first step into running an emulator on your computer within Wylodrin Studio is to install the right version of the QEMU machine emulator for your computer.

Install QEMU for [Linux](#).

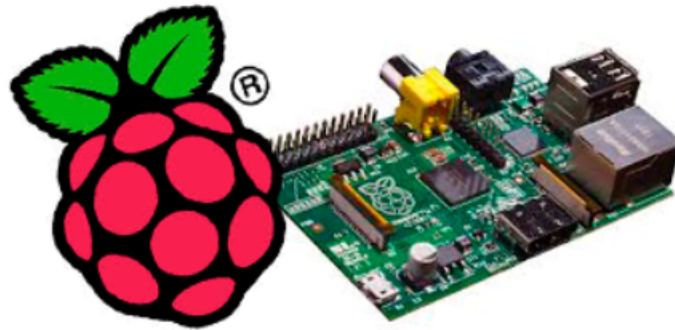
Install QEMU for [Windows](#).

If your PC is running on Windows, you will have to add qemu in the PATH variable. In order to accomplish that, you will have to right click on *This PC*, select *Properties*, open the *Environment Variables* option, then edit the *PATH* variable. Here, you will have to add the absolute path to the folder where you chose to install qemu. The last step is to save the changes.

Install QEMU for [mac OS](#).

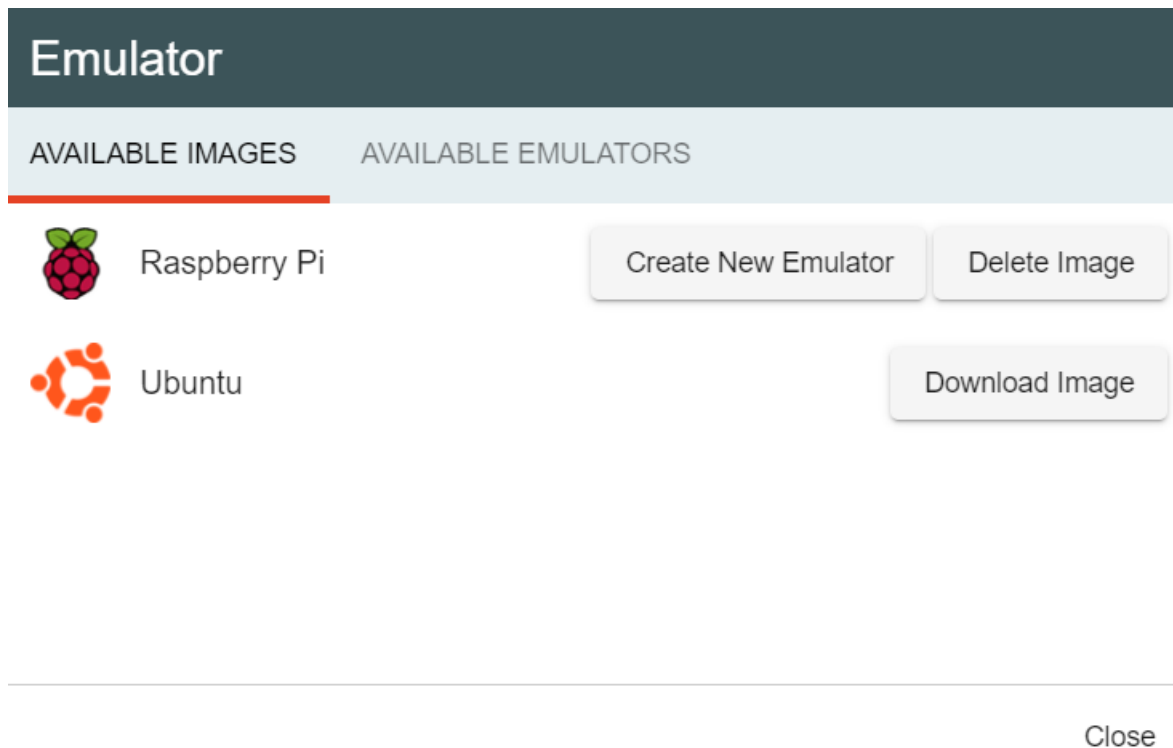
Compile the QEMU [source code](#).

10.1.2 Raspberry Pi Emulator



Once you have the QEMU machine installed on your computer, you will be able to emulate a Raspberry Pi within Wylodrin STUDIO. If you open the IDE, you will find the Emulator option between one of the items of the Menu.

If you don't have any emulator previously created on your computer, the first tab will be automatically displayed in the prompt that will pop up. Here, you will be able to see a list with all the supported types of emulators.



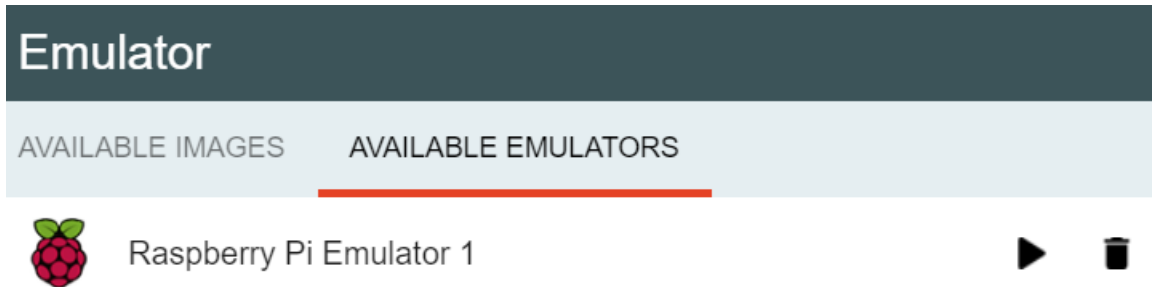
If you don't have any kernel image, you will have the option to download one previously configured by us. By clicking on the "Download image" button, a zip archive will be downloaded and unzipped in a special folder created on your computer. Once the download and decompression processes will be done, 2, new options will be available for the emulator.

As you can see in the picture shown above, you can either click on the **"Delete image"** button, that will permanently remove the kernel image from your computer, or on the **"Create new emulator"** button.

This last option will pop up a prompt where you will be asked to input the name of your emulator. You will have to enter a valid name, having at least one character, that has not been already used for another emulator. This action will

start the boot process for your emulator, by copying the kernel image into a folder specifically created for the new emulator. In a few minutes, the Raspberry Pi emulator will show up on your computer.

By switching to the second tab of the Emulator prompt, you will see a list of all the available emulators that exist on your computer.



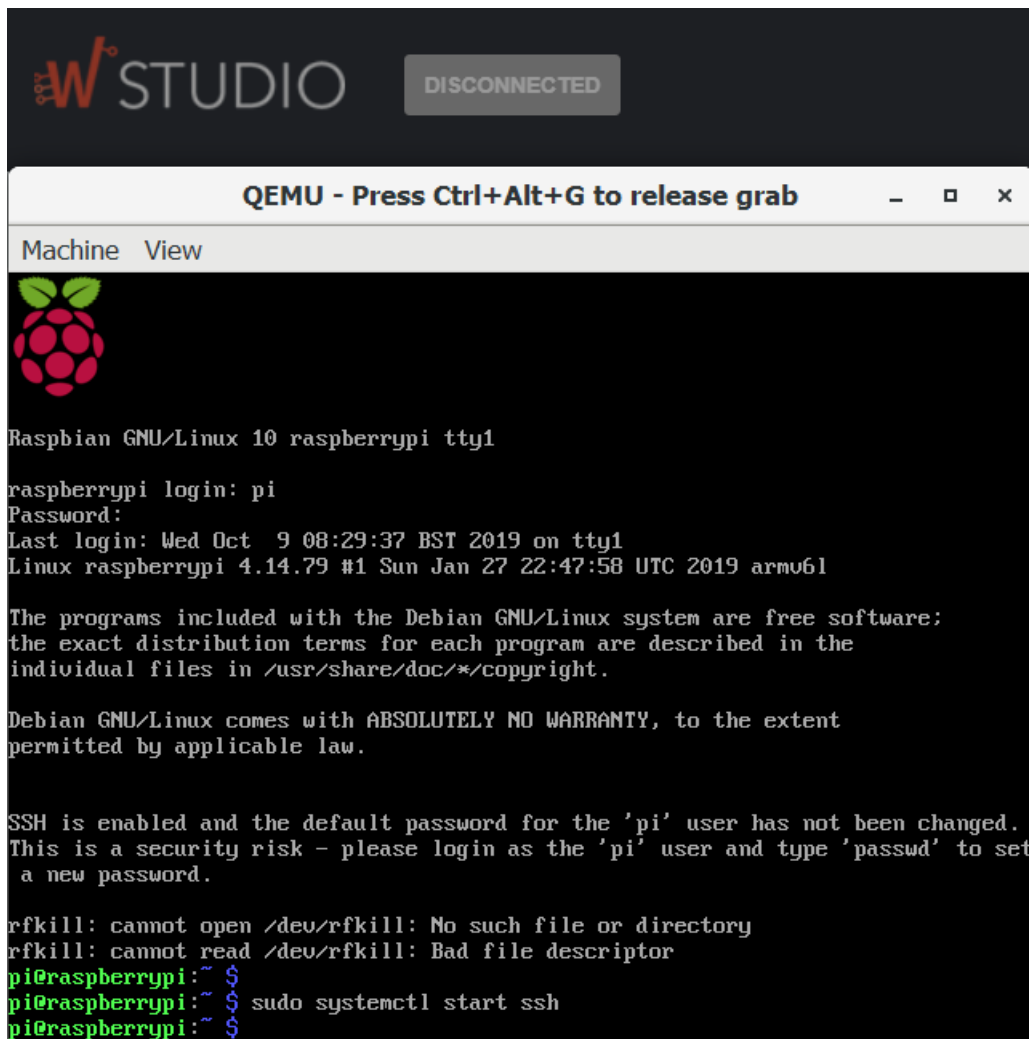
Close

Here, for each available emulator, you will have 3 options:

- **Stop Emulator** - this button will kill the session for a chosen emulator, but your settings will be saved within the special folder.
- **Restart Emulator** - this button will be visible on each restart of Wylodrin STUDIO or after each stop of an emulator. It will allow you to restart an emulator and to reload your changes and settings.
- **Delete Emulator** - this button will ask you if you really want to delete an emulator. By answering *yes*, the selected emulator will be permanently removed from your computer and you will lose all the saved data.

Connect to the emulator

Once the emulator completely loaded, you will be asked to input the default username and password, which are: *pi / raspberry*. After that, you will have to start the ssh session by typing: *sudo systemctl ssh start*



After that, you will be able to find your emulator in the Connection Menu and to connect to it.

11.1 RaspberryPi Simulator

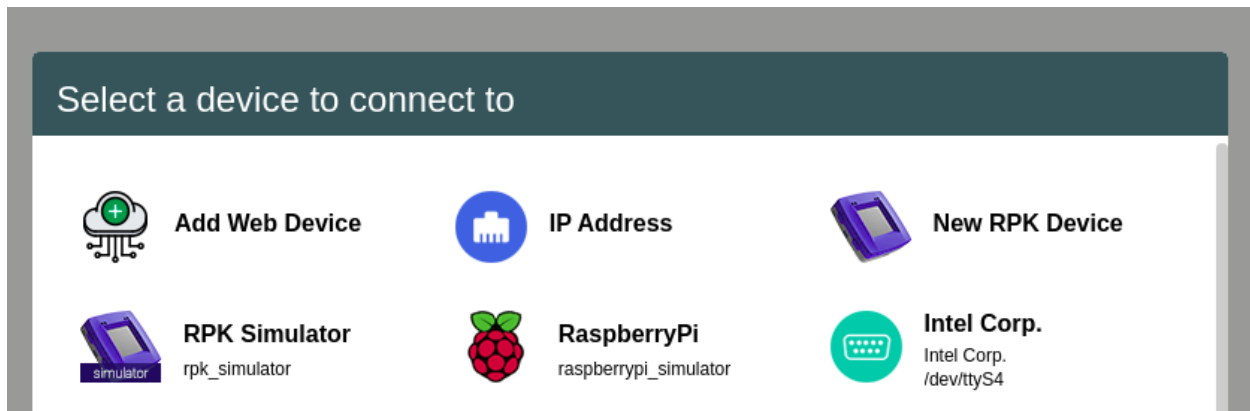
The **RaspberryPi Simulator** is a plugin used for the simulation of simple circuits using a RaspberryPi. It can simulate series circuits made of leds, buttons and LCDs. The code to be runned on the RaspberryPi is written in the **Application** tab, and the only supported language is **NodeJS**. So, the basic 2 components of this plugin is the **project** and the **schema** of the circuit.

11.1.1 Steps to use the RaspberryPi Simulator

You can use the RaspberryPi Simulator by following the next steps.

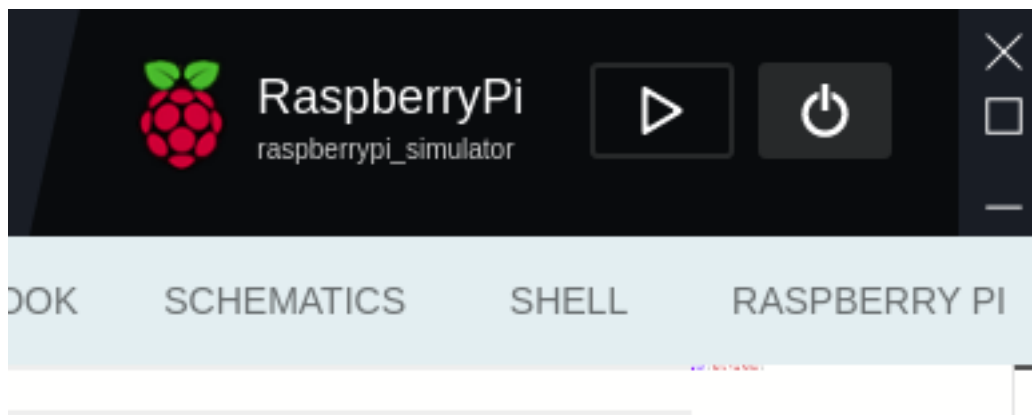
1. Connection

If you are connected to another device, disconnect from it. After that, press the **CONNECT** button in the headbar, and afterwards choose the “**RaspberryPi**” that runs on the board **raspberrypi_simulator**. This simulates a connection to a physical board.

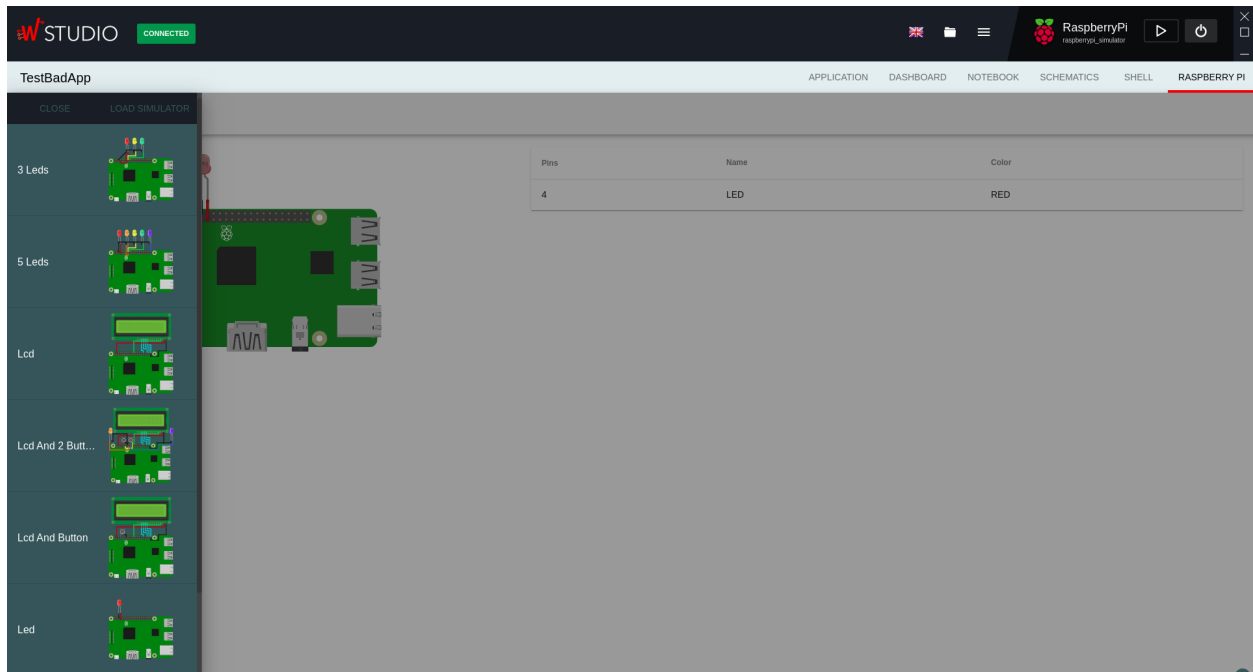


2. RaspberryPi Simulator tab

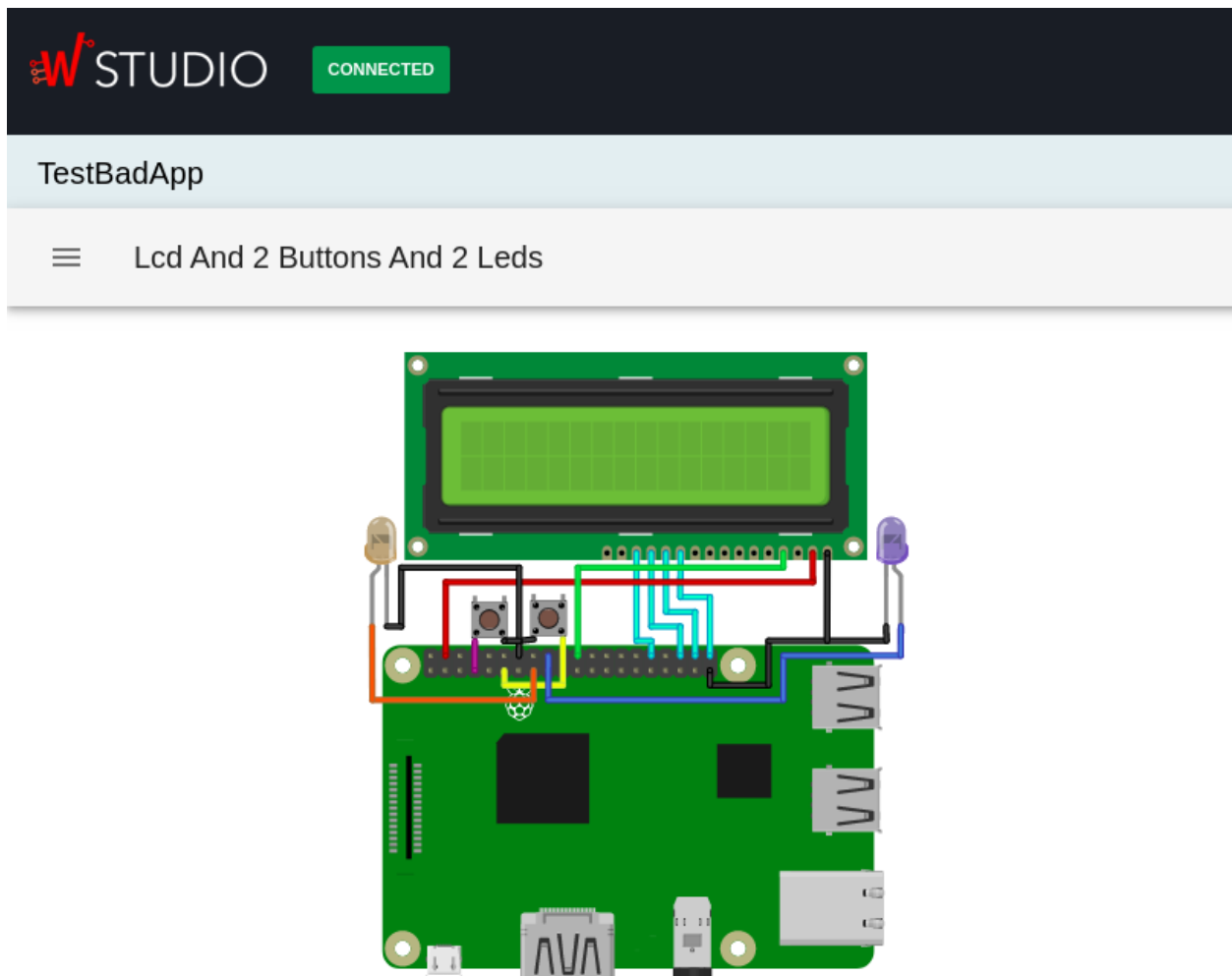
After you connect to the RaspberryPi Simulator board, a new tab will appear on the right in the tab list, named **RASPBERRY PI**. This tab has a structure of three components:



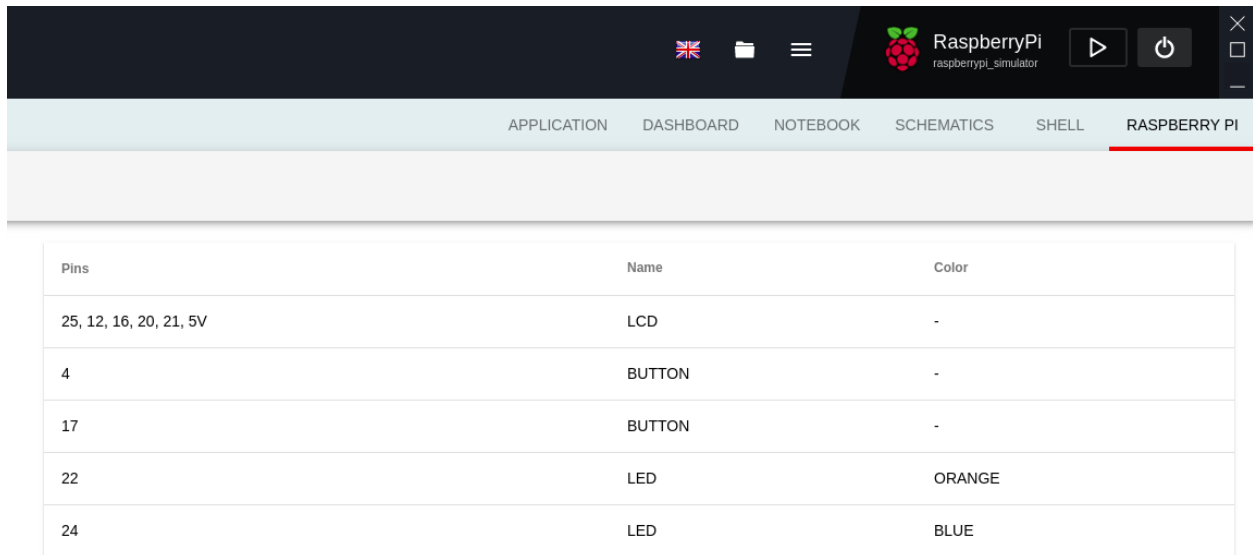
1. **The button to open the list of examples** It is located right under the tab list. After you press the button, a list of examples will be opened on the left side. At the top of the list will be a button **LOAD SCHEMA** used to load your own schema. We will discuss about that later.



2. **The circuit image** It is located on the left side of the window. The image is interactive, meaning that the leds can turn on, the buttons can be pressed, and so on.



3. **The table of connections** It is located on the right side of the window. Here you will see a simplified structure of the connections, so you can see which pins are connected in order to know how to use the components.



3. Load a project and run it

After you got accustomed to the **RaspberryPi Simulator tab**, you can load a project written in **NodeJS**. There are only 2 libraries available at the moment, in order to control the GPIO pins on the RaspberryPi and the LCDs.

Afterwards, you can just press the **RUN** button and the code will start to execute. Also, the console remains active in the RaspberryPi Simulator tab, so you can see the evolution of your code while interacting with the schema loaded.

11.1.2 Load your own schema

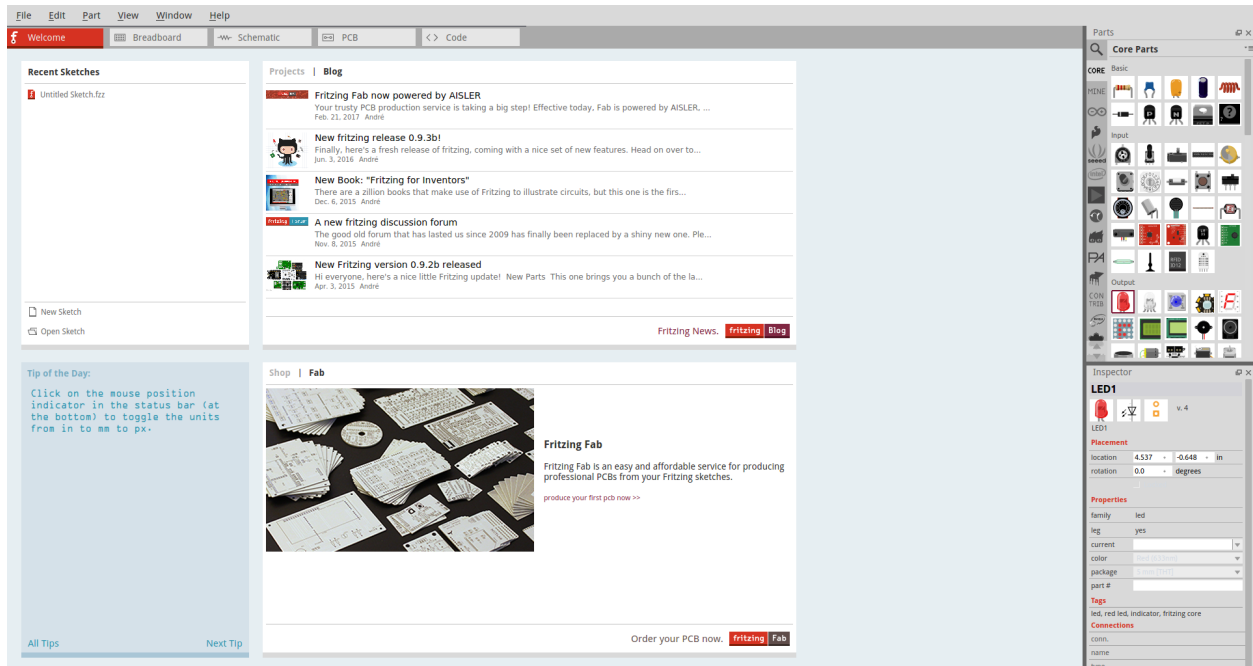
The RaspberryPi Simulator plugin runs on schemas created with Fritzing. A schema has 2 components:

- the image of the circuit, saved in a **SVG** format
- the **netlist** with the connections, saved in a **XML** format

In order to create your own schema, you can follow the next steps.

1. Download and open Fritzing

You can access the Fritzing download page by clicking [here](#). After you download the application, open it. You will see the below window.



2. Add components

After you open the app, you have to go to the **Breadboard tab**. From this tab you will export the files required for the schema, meaning the **SVG** image and the **XML** netlist. On the right-top side you will see a window with components. From there you can search for the desired the components. The available components that are recognized by the simulator are:

- **RaspberryPi 3**
- **Pushbutton**
- **LED**
- **LCD 16x2**

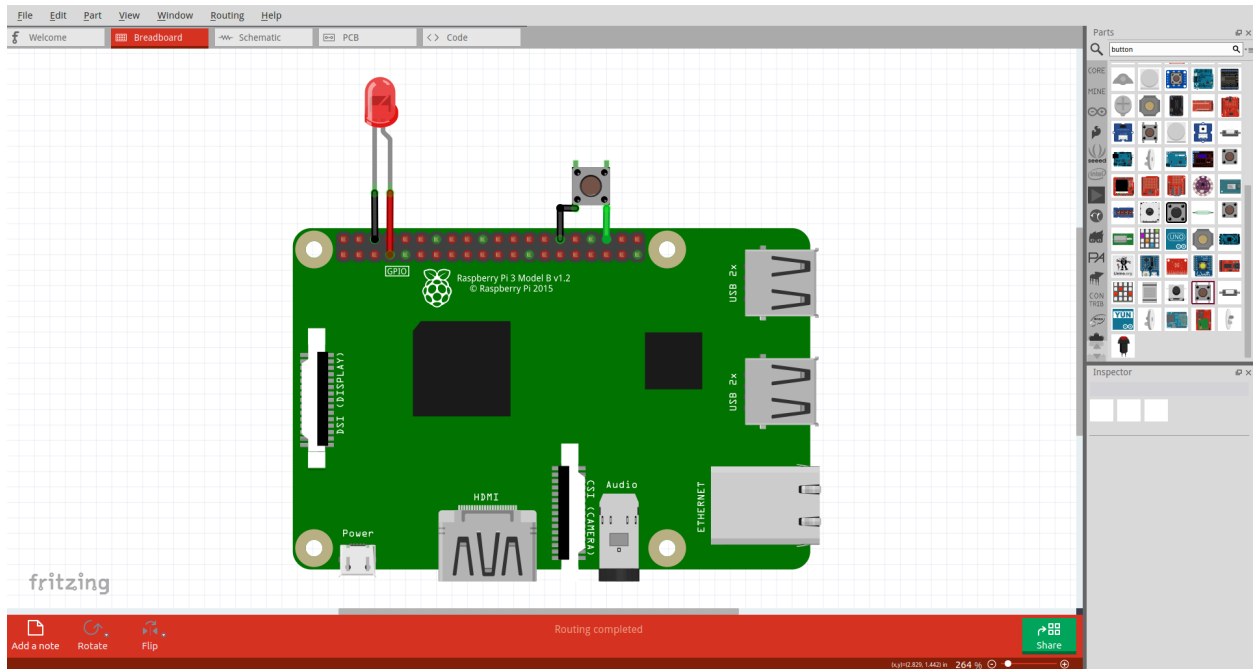
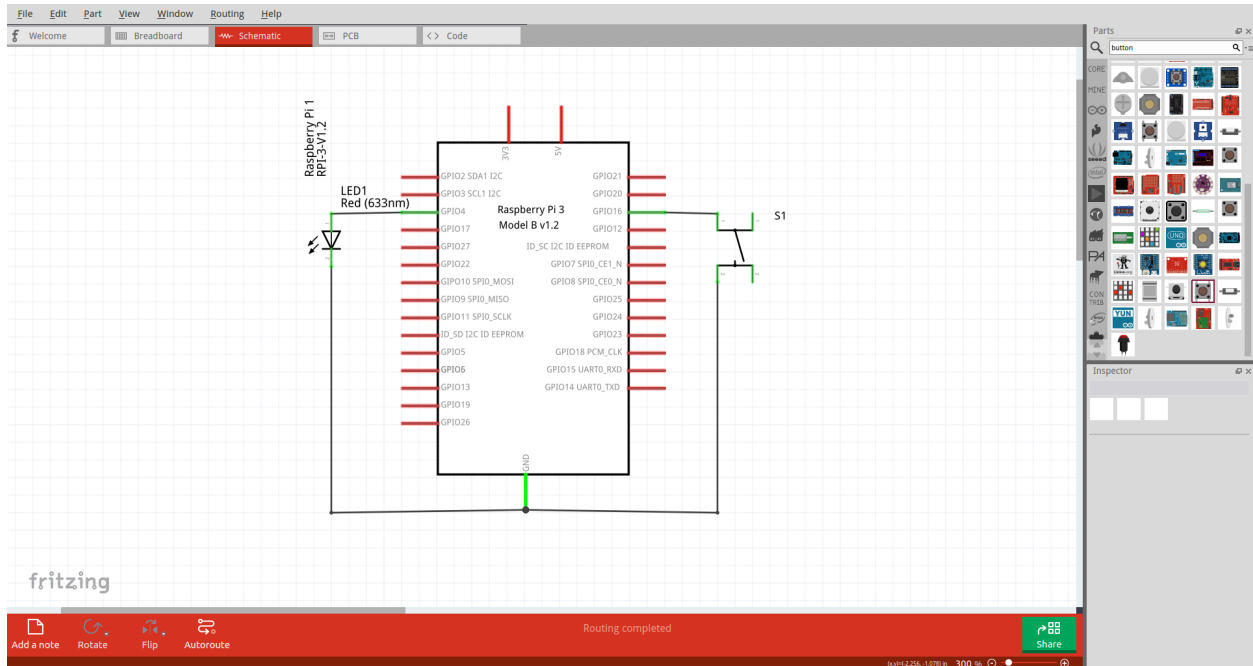
In order to place a component with just have to drag and drop it on the main panel from the Breadboard tab.



3. Make connections

After you've placed all the required components, you have to access the **Schematic tab**. There you will see all components and you can make the connections between them. After the connections are done, you have to go back on the **Breadboard tab** and make the physical connections again. There will be dotted lines that correspond to the connections from the Schematic tab. Also, you can edit the components from the window on the right-bottom. For

example, you can change the color of the led, or the color of the wire.



4. Export from Fritzing

Attention! In order to export the files from the schema created, you have to be on the Breadboard tab.

To export the required files for the schema, you have to follow 2 steps:

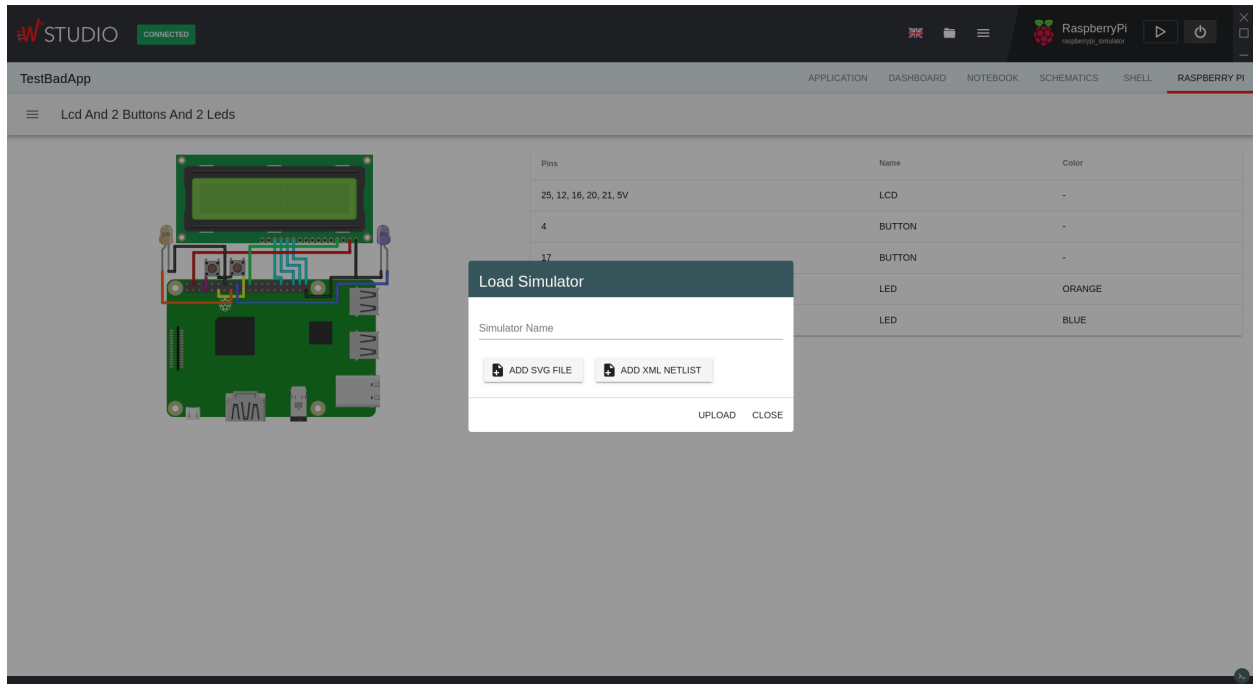
- export the SVG file: **File -> Export -> as Image -> SVG...**
- export the XML netlist: **File -> Export -> XML Netlist...**

You have to save the files on your computer in an easy accesible location, because you will need the afterwards.

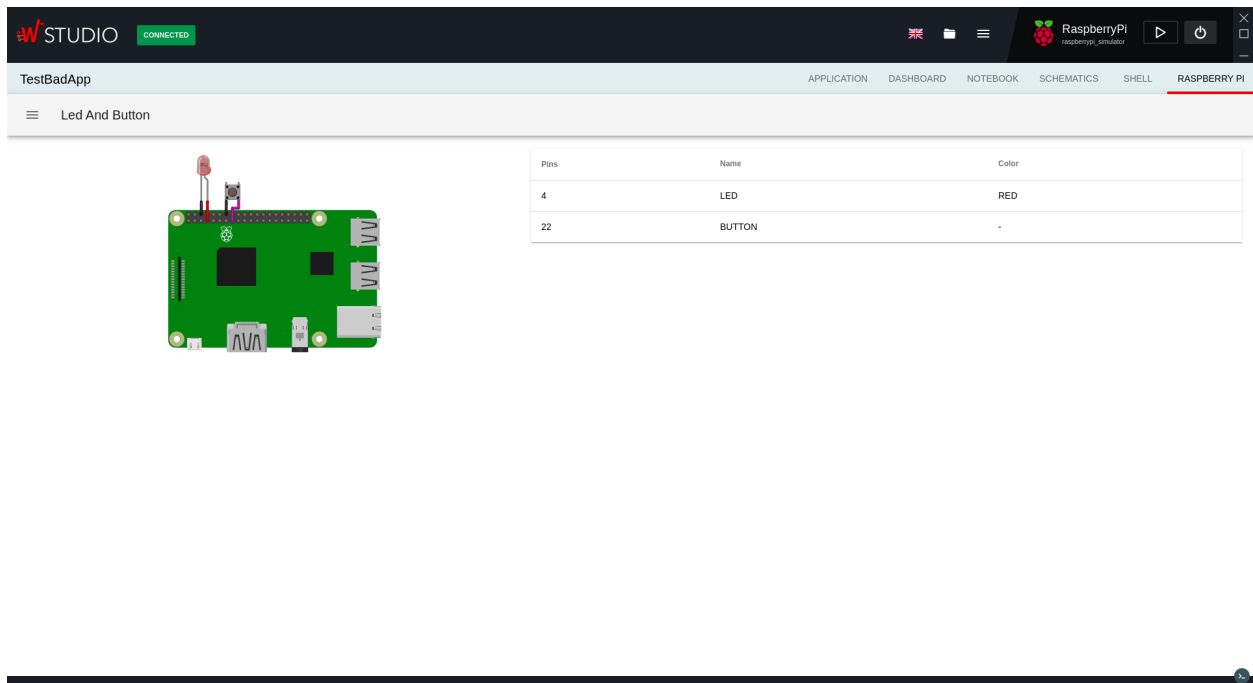
5. Import to Wylodrin STUDIO

To import the files just created with Fritzing, you have to follow the next steps:

- access the **RASPBERRY PI** tab
- click on the button to open the list with examples
- press **LOAD SCHEMA**, and a pop-up will appear on your screen
- give a name to your schema if you want
- press **ADD SVG FILE** and choose your just-created SVG file
- press **ADD XML NETLIST** and choose your just-created XML netlist
- press **UPLOAD**



If all the steps are followed correctly, your schema should appear on the main view, along side with the table of connected components.



11.1.3 Libraries for RaspberryPi Simulator

In the RaspberryPi Simulator you can almost fully use two main libraries: **onoff.GPIO** and **LCD**. The usage of these libraries is the same as the usage on phisical boards. The only difference is that not every function is available. Bellow you can see a list of available function for the 2 libraries. To see more of the usage for these libraries you can access one of the links bellow:

- [onoff](#)
- [LCD](#)

onoff.GPIO:

- **create(pin, state)** -> creates an object from which you can control the GPIO pins on the RaspberryPi. The *state* parameter is a string and it indicates the type for the pin **input/output**
- **readSync()** -> return the value readed by the pin **1/0**
- **writeSync(value)** -> outputs on the GPIO pin the selected value **1/0**
- **direction()** -> return the state of the pin
- **setDirection(state)** -> change the state of the pin
- **activeLow()** -> return the state of the *activeLow* property of the pin
- **setActiveLow(value)** -> change the activeLow property on the pin

LCD:

- **create(object)** -> it creates an object in order to interact with the LCD. The object contains 4 properties: *rs*, *e*, *data*, *cols*, *rows*
- **print(string)** -> print the given string on the screen starting from the cursor current position

- **clear()** -> clears the LCD screen
- **home()** -> sets the cursor on the cell **0x0** on the LCD
- **setCursor(row, col)** -> sets the cursor on the *row* line and *col* column
- **scrollDisplayLeft()** -> scrolls the display one position to the left
- **scrollDisplayRight()** -> scrolls the display one position to the right
- **close()** -> close the connection with the LCD and free the assigned pins

Attention! LCD library only supports the 16x2 LCD!

11.1.4 Code examples

Bellow are 2 code examples on how to use the **onoff.GPIO** library and the **LCD** library.

onoff.GPIO example

The schematic associated to this code should have a led connected to *GPIO4* pin on the RaspberryPi and to the *GND* pin, and a button connected to the *GPIO22* pin and to the *3V* pin.

This program will wait 2 seconds, will turn the led on, then will wait another 2 seconds and will turn it off. Afterwards, it will remain in the *while* loop while the button is not pressed.

```
var Gpio = require("onoff").Gpio;

var led = new Gpio(4, "out");
var button = new Gpio(22, "in");

sleep(2000);
led.writeSync(1);
sleep(2000);
led.writeSync(0);

while(button.readSync() === 0) {
    sleep(1000);
}

console.log("onoff.Gpio tutorial finished!");
```

LCD example

The schematic associated to this code should have a LCD connected as it follows:

- VSS connected to the *GND* pin

- VDD connected to the 5V pin
- RS connected to the *GPIO25* pin
- E connected to the *GPIO2* pin
- DB4 connected to the *GPIO23* pin
- DB5 connected to the *GPIO17* pin
- DB6 connected to the *GPIO18* pin
- DB7 connected to the *GPIO22* pin

This program will write the “Hello world, from the LCD!” string on the LCD. Because it won’t fit, we will scroll the display to the left for 10 times, and then we will clear the display. At the end, we close the connection to the LCD.

```
var LCD = require("lcd");
var lcd = new LCD({rs: 25, e: 2, data: [23, 17, 18, 22], cols: 16, rows: 2})

lcd.print("Hello World, from the LCD!");

sleep(2000);

for (var i = 0; i < 10; i ++) {
    sleep(1000);
    lcd.scrollDisplayLeft();
}

sleep(1000);
lcd.clear();

lcd.close();

console.log("LCD tutorial finished!");
```

C

changeFile() *(built-in function)*, 85
cloneProject() *(built-in function)*, 80
closeStatusButton() *(built-in function)*, 72
connect() *(built-in function)*, 74
createEmptyProject() *(built-in function)*, 79

D

deleteFile() *(built-in function)*, 84
deleteFolder() *(built-in function)*, 87
deleteProject() *(built-in function)*, 80
Device() *(class)*, 67
disconnect() *(built-in function)*, 75
dispatchToStore() *(built-in function)*, 73
disposable() *(built-in function)*, 67, 77

E

exportProject() *(built-in function)*, 81

F

file() *(class)*, 77

G

generateStructure() *(built-in function)*, 83
getCurrentFileCode() *(built-in function)*, 87
getCurrentProject() *(built-in function)*, 83
getDefaultFileName() *(built-in function)*, 86
getDefaultRunFileName() *(built-in function)*, 86
getDevice() *(built-in function)*, 75
getFileCode() *(built-in function)*, 86
getFromStore() *(built-in function)*, 72
getMakefile() *(built-in function)*, 86
getStatus() *(built-in function)*, 75

I

importProject() *(built-in function)*, 81

L

Language() *(class)*, 76

languageSpecificOption() *(built-in function)*, 79

loadFile() *(built-in function)*, 84
loadPreviousSelectedCurrentProject() *(built-in function)*, 82
loadProjects() *(built-in function)*, 82
loadSettings() *(built-in function)*, 95
loadSpecialFile() *(built-in function)*, 85
loadValue() *(built-in function)*, 95

N

newFile() *(built-in function)*, 83
newFolder() *(built-in function)*, 87

O

openStatusButton() *(built-in function)*, 71

P

Project() *(class)*, 76
projects.getLanguage() *(projects method)*, 77

R

recursiveCreating() *(built-in function)*, 81
recursiveGeneration() *(built-in function)*, 82
registerComponent() *(built-in function)*, 73
registerDeviceDriver() *(built-in function)*, 74
registerDeviceToolButton() *(built-in function)*, 57, 70
registerEditor() *(built-in function)*, 78, 109
registerLanguage() *(built-in function)*, 77
registerLanguageAddon() *(built-in function)*, 78
registerMenuItem() *(built-in function)*, 53, 69
registerPinLayout() *(built-in function)*, 89
registerStatusButton() *(built-in function)*, 58, 71
registerStore() *(built-in function)*, 72
registerTab() *(built-in function)*, 56, 68
registerToolBarButton() *(built-in function)*, 55
renameMenuItem() *(built-in function)*, 69

`renameObject()` (*built-in function*), 87
`renameProject()` (*built-in function*), 80

S

`saveFile()` (*built-in function*), 84
`saveSpecialFile()` (*built-in function*), 85
`selectCurrentProject()` (*built-in function*), 82
`setWorkspaceTitle()` (*built-in function*), 73
`showConfirmationPrompt()` (*built-in function*),
123
`showConnectionSelectionDialog()` (*built-in
function*), 76
`showDialog()` (*built-in function*), 119
`showError()` (*built-in function*), 124
`showNotification()` (*built-in function*), 123
`showPrompt()` (*built-in function*), 122
`storeSettings()` (*built-in function*), 94
`storeValue()` (*built-in function*), 95

U

`updateDevices()` (*built-in function*), 74