Table of Contents

About

1. Getting Started

2. Pymakr Plugin

3. Pysense & Pytrack

4. Tutorials & Examples

5. Firmware & API Reference

6. Product Info

7. Datasheets

8. Pybytes

9. Documentation Notes

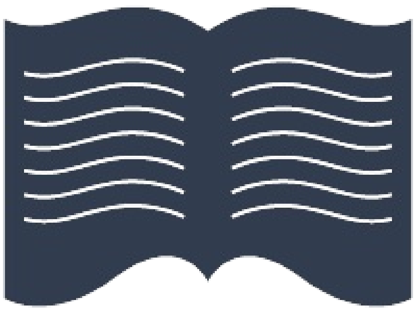
10. Advanced Topics

11. License

Pycom Documentation

Welcome to the Pycom documentation site. The documentation is split into 5 sections; we recommend reading through all the sections to familiarise yourself with the various tools and features available to you to help you develop on your Pycom module.

To get started, read through the Getting Started Guide then feel free to jump straight into the tutorials and examples in Tutorials & Examples to begin building your projects.



Products

Getting Started

Tutorials

Product Info

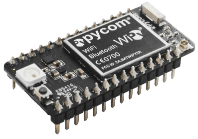
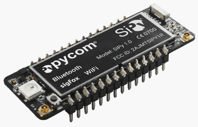
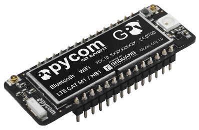
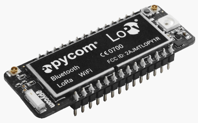
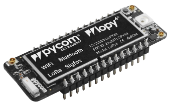
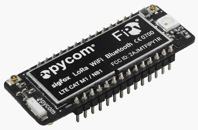
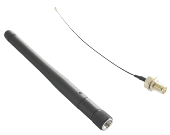
API Documentation

Pybytes

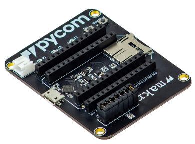
Pycom Products

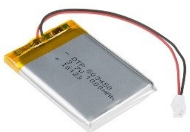
Below you will find tables of all Pycom products. These tables illustrate the functionality of our various products, their compatibility with each other, as well as what accessories are required to utilise certain functionality.

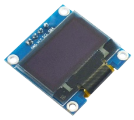
Development Boards

Module	WiFi	Bluetooth	LoRa	Sigfox	LTE CAT-M1 NB-IoT
 WiPy 3.0	✔	✔			
 SiPy	✔	✔		✔	
 GPy	✔	✔			✔
 LoPy	✔	✔	✔		
 LoPy4	✔	✔	✔	✔	
 FiPy	✔	✔	✔	✔	✔
Antennas	 External WiFi/BT Antenna Kit		 LoRa & Sigfox Antenna Kit		 LTE-M Antenna Kit

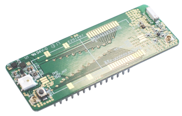
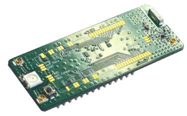
Accessories

Accessory	 **Expansion**			

	Board	Pysense	Pytrack	Pyscan
 PyCase	✔			
 IP67 Case for Expansion Board	✔			
 IP67 Case for Pysense/Pytrack/Pyscan		✔	✔	✔
 IP67 Case (universal)	✔	✔	✔	✔
 LiPo Battery (user-supplied)	✔	✔	✔	✔
 Micro USB Cable *Required* (user-supplied)	✔	✔	✔	✔

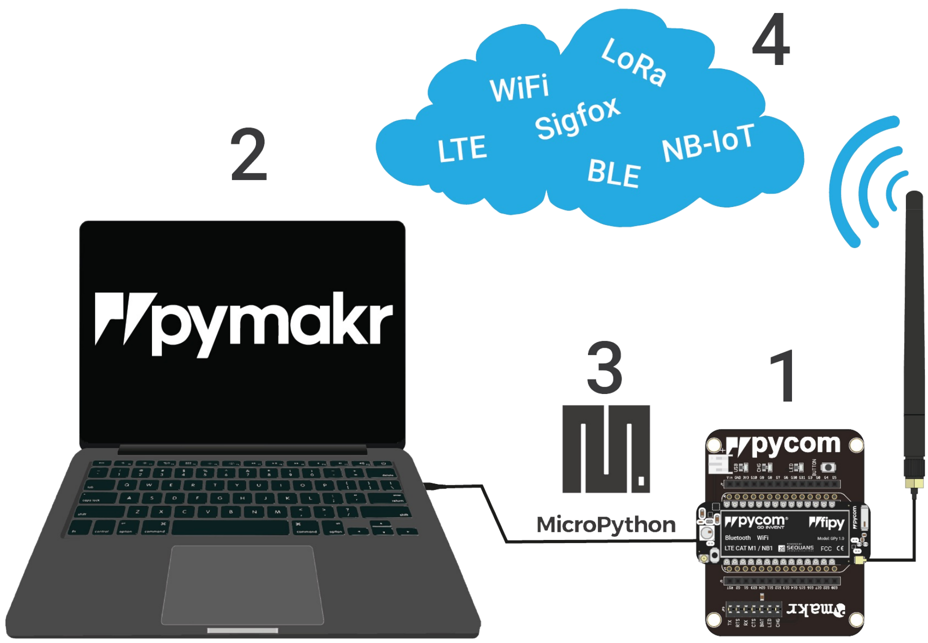
Pyscan Modules	 OLED Module  2MP Camera  Barcode Reader  Fingerprint Scanner  IR Image Sensor	✔

OEM Modules

OEM Module	 L01/W01 Reference Board	 Universal Reference Board
 W01	✔	✔
 L01	✔	✔
 L04		✔
 G01		✔

Getting Started

So, you've decided to order a Pycom development module. Firstly we would like to congratulate you in making an excellent decision. If you haven't yet placed your order we highly recommend you check out the products page before you place your order to ensure you know which accessories you might require.



Step 1: Setting up the hardware

In the first part of this getting started guide, we will take you through setting up your device. Firstly we will cover how to connect the module to your computer either via USB or WiFi. Secondly we will explain how to connect various accessories such as antennas or SIM cards to your module.

Step 2: Setting up your computer

Now that your module is successfully connected, you will need to install some software on your computer to interface with it. The second part of this guide will guide you through installing drivers; performing firmware updates for your module/accessories to ensure you have the most stable and feature packed version; and how to setup the software use to program the device.

Step 3: Using your module

Now that you have a connected module and all the required software installed it is time to begin programming your device. This part of the guide will get you started with a basic example and point you in the right direction for getting your device connected to your chosen network.

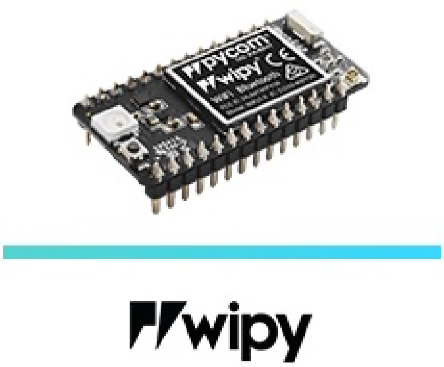
Step 4: Connecting to a network

Now that you familiar with programming your device you will no doubt be keen to get it connected to one of the advertised wireless networks. This usually requires some registration. This step will detail how to get registered and connected to various wireless networks.

> You can navigate through this guide using the arrow buttons on the left and right of the screen (or at the bottom if you are using mobile).

Setting up the hardware

This chapter of the documentation will show you how to connect you Pycom module. For each device there are detailed instructions on how to connect your module to one of our base boards, a USB UART adapter or WiFi as well as what antennas you might need to connect. Please select your module below to be taken to the appropriate guide.




lopy4


sipy


gpy

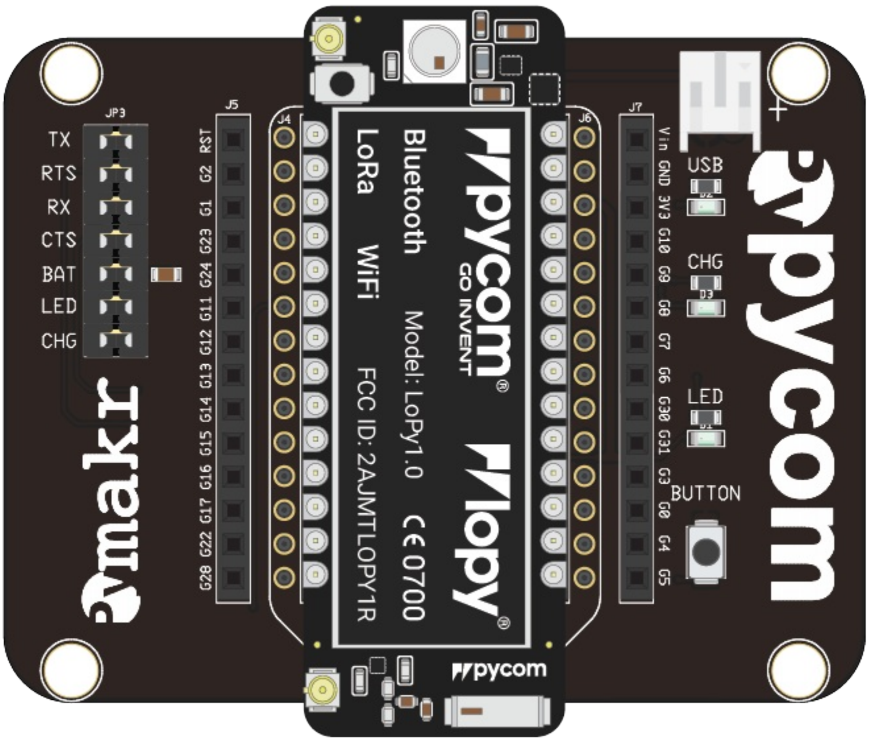

pycom
fipy
WiFi Bluetooth LoRa
Sigfox LTE CAT M1 / NB1

LoPy

Basic connection

- Exp Board 2.0
- Exp Board 3.0
- Pysense/Pytrack/Pyscan
- USB UART Adapter
- WiFi

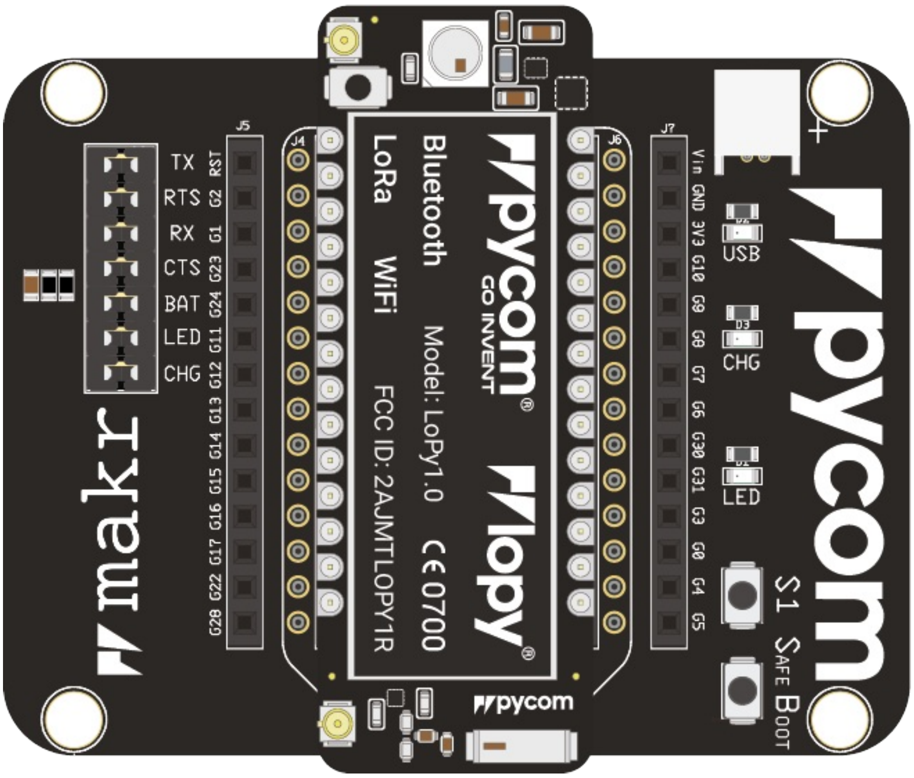
- Look for the reset button on the module (located at a corner of the board, next to the LED).
- Locate the USB connector on the expansion board.
- Insert the LoPy module on the the expansion board with the reset button pointing towards the USB connector. It should firmly click into place and the pins should now no longer be visible.



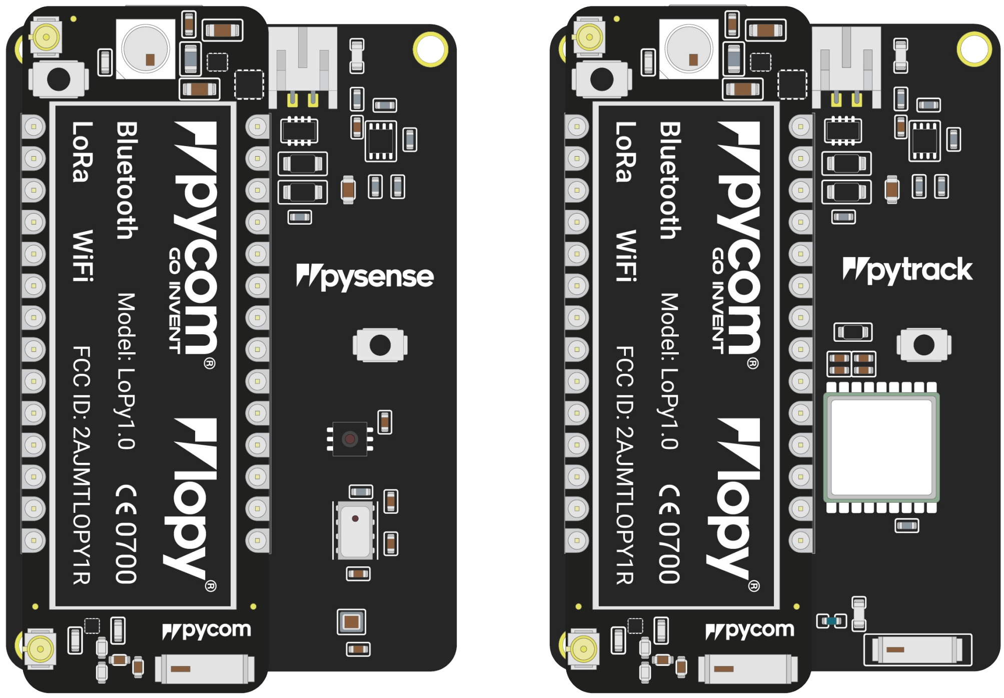
- Before connecting your module to an Expansion Board 3.0, you should update the firmware on the Expansion Board 3.0. Instructions on how to do this can be found here.

- Look for the reset button on the module (located at a corner of the board, next to the LED).

- Locate the USB connector on the expansion board.
- Insert the LoPy module on the Expansion Board with the reset button pointing towards the USB connector. It should firmly click into place and the pins should now no longer be visible.

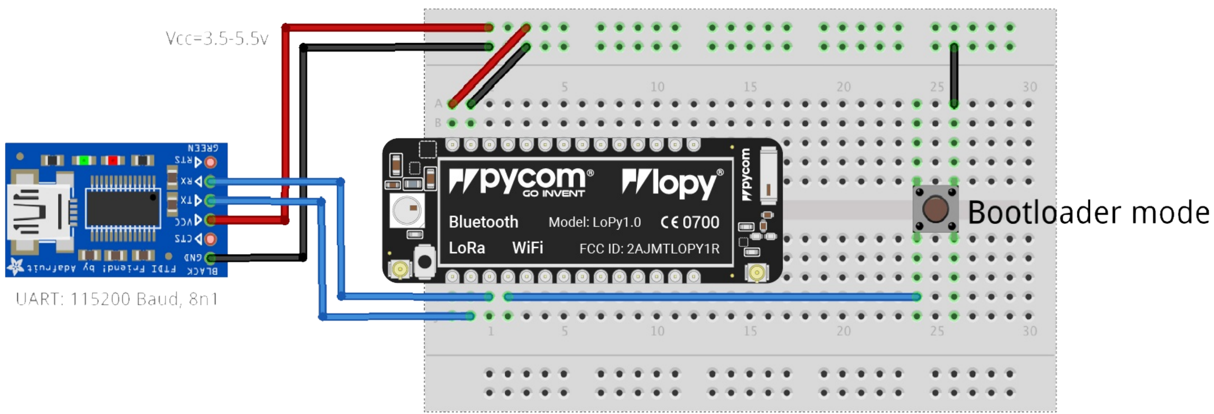


- Before connecting your module to a Pysense/Pytrack/Pyscan board, you should update the firmware on the Pysense/Pytrack/Pyscan. Instructions on how to do this can be found here.
- Look for the reset button on the LoPy module (located at a corner of the board, next to the LED).
- Locate the USB connector on the Pysense/Pytrack/Pyscan.
- Insert the module on the Pysense/Pytrack/Pyscan with the reset button pointing towards the USB connector. It should firmly click into place and the pins should now no longer be visible.



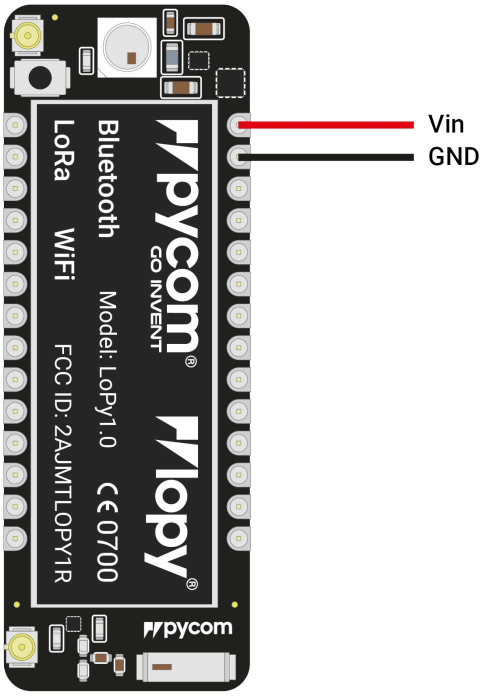
Once you have completed the above steps successfully you should see the on-board LED blinking blue. This indicates the device is powered up and running.

- Firstly you will need to connect power to your LoPy. You will need to supply `3.5v` - `5.5v` to the `Vin` pin. **Note:** Do *not* feed `3.3v` directly to the `3.3v` supply pin, this will damage the regulator.
- The connect the `RX` and `TX` of your USB UART to the `TX` and `RX` of the LoPy respectively. **Note:** Please ensure you have the signal level of the UART adapter set to `3.3v` before connecting it.
- In order to put the LoPy into bootloader mode to update the device firmware you will need to connect `P2` to `GND`. We recommend you connect a button between the two to make this simpler.



Note: This method of connection is not recommended for first time users. It is possible to lock yourself out of the device, requiring a USB connection.

- In order to access the LoPy via WiFi you only need to provide `3.5v` - `5.5v` on the `Vin` pin of the LoPy:



- By default, when the LoPy boots, it will create a WiFi access point with the following credentials:
    - SSID: `lopy-wlan`
    - password: `www.pycom.io`
    - Once connected to this network you will be able to access the telnet and FTP servers running on the LoPy. For both of these the login details are:
    - username: `micro`
    - password: `python`

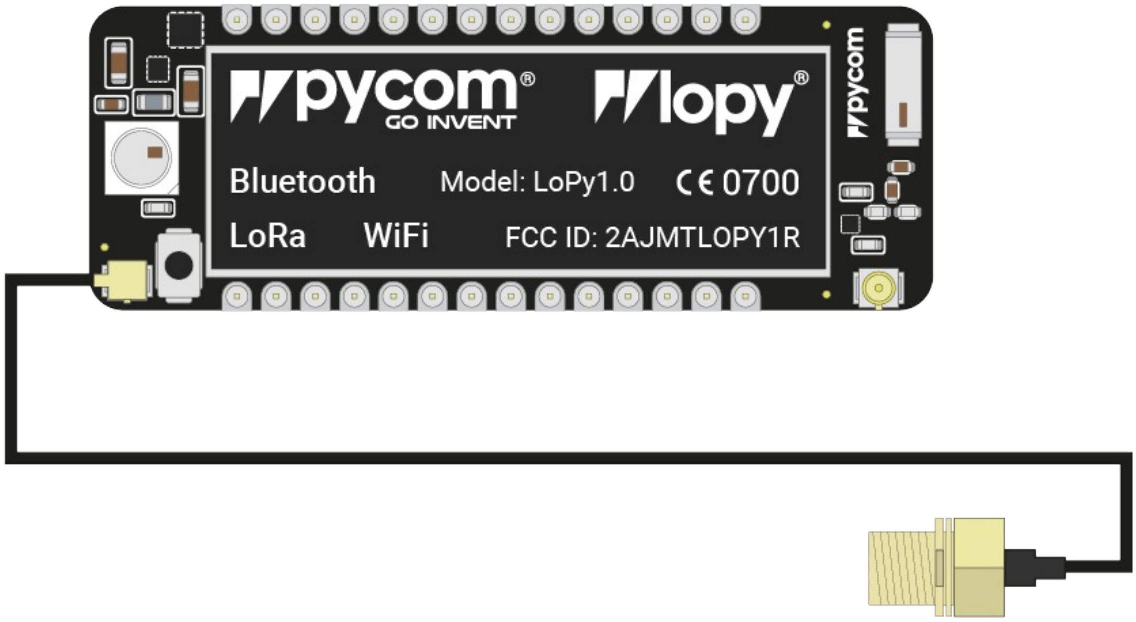
Antennas

LoRa

> If you intend on using the LoRa connectivity of the LoPy you **must** connect a LoRa antenna to your LoPy before trying to use LoRa otherwise you risk damaging the device.

The LoPy only supports LoRa on the 868MHz or 915MHz bands. It does not support 433MHz. For this you will require a LoPy4.

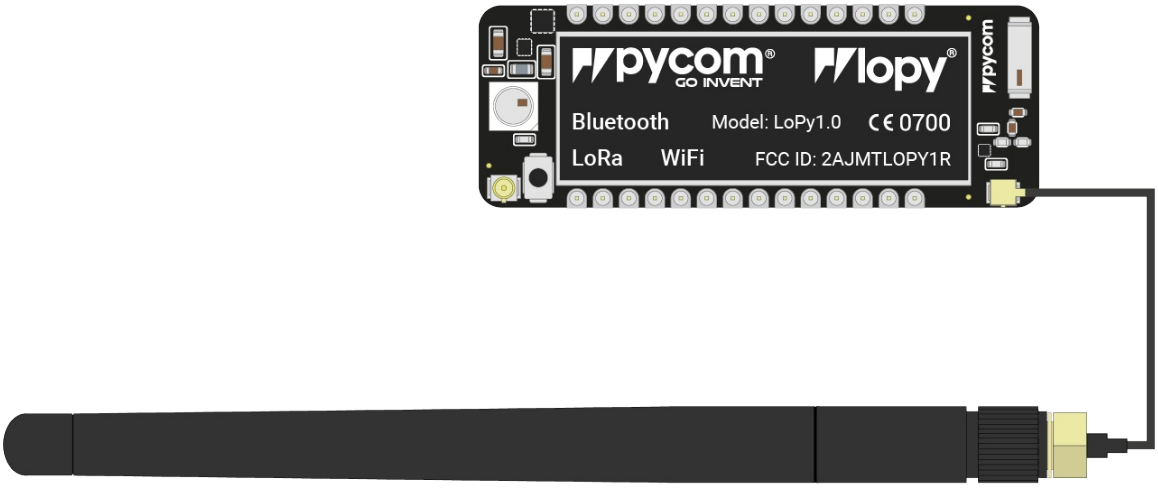
- Firstly you will need to connect the U.FL to SMA pig tail to the LoPy using the U.FL connector on the same side of the LoPy as the LED.



- If you are using a pycase, you will next need to put the SMA connector through the antenna hole, ensuring you align the flat edge correctly, and screw down the connector using the provided nut.
- Finally you will need to screw on the antenna to the SMA connector.

WiFi/Bluetooth (optional)

All Pycom modules, including the LoPy, come with a on-board WiFi antenna as well as a U.FL connector for an external antenna. The external antenna is optional and only required if you need better performance or are mounting the LoPy in such a way that the WiFi signal is blocked. Switching between the antennas is done via software, instructions for this can be found here.



Deep Sleep current issue

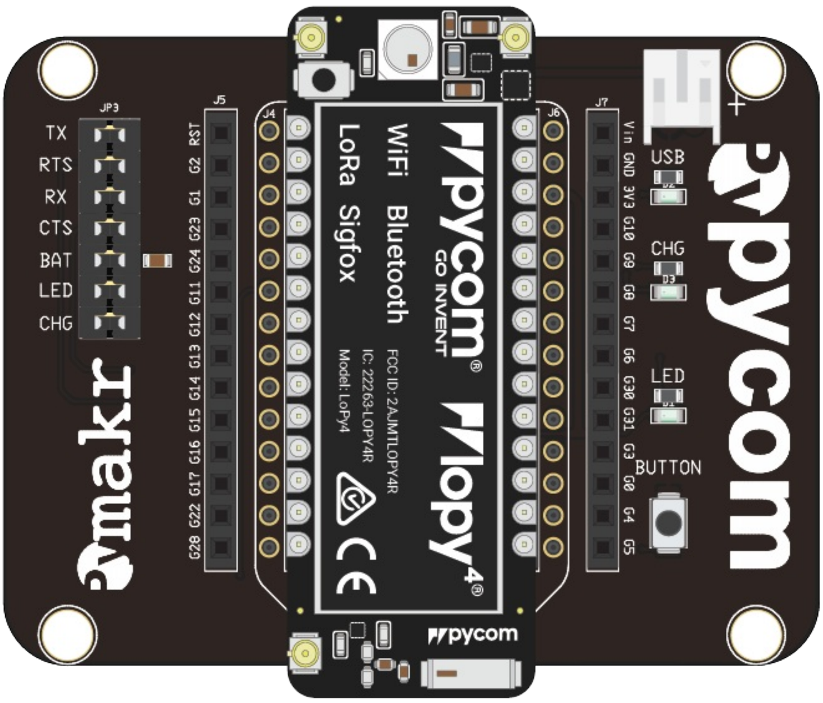
The LoPy, SiPy, and WiPy 2.0 experience an issue where the modules maintain a high current consumption in deep sleep mode. This issue has been resolved in all newer products. The cause for this issue is the DC to DC switch mode converter remains in a high performance mode even when the device is in deep sleep. The flash memory chip also does not power down. A more detailed explanation can be found here.

LoPy4

Basic connection

- Exp Board 2.0
- Exp Board 3.0
- Pysense/Pytrack/Pyscan
- USB UART Adapter
- WiFi

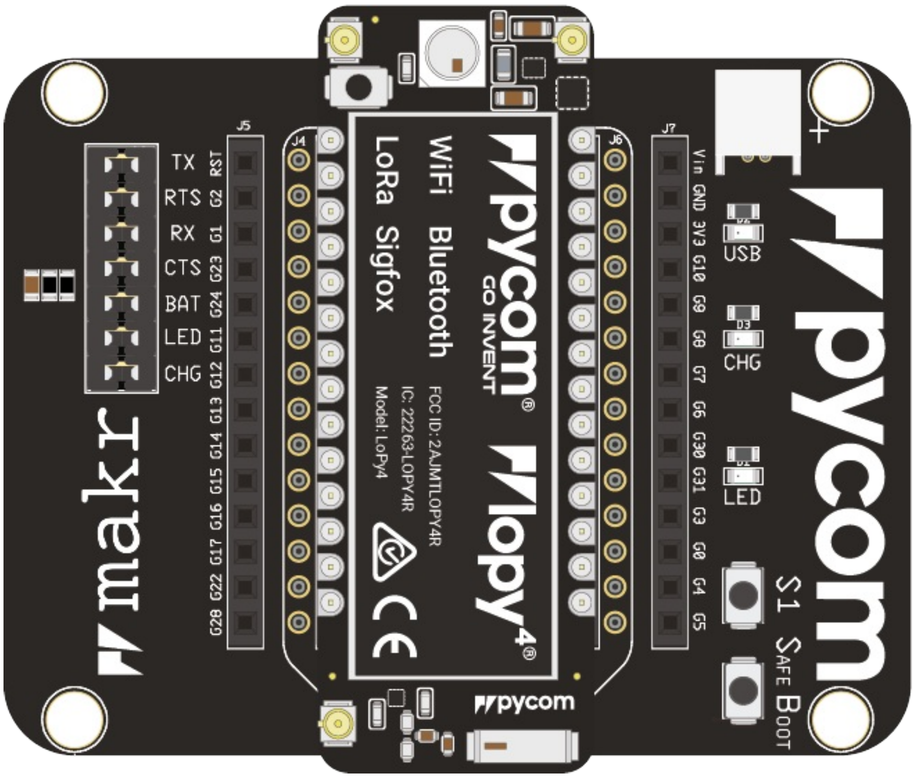
- Look for the reset button on the module (located at a corner of the board, next to the LED).
- Locate the USB connector on the expansion board.
- Insert the LoPy4 module on the the expansion board with the reset button pointing towards the USB connector. It should firmly click into place and the pins should now no longer be visible.



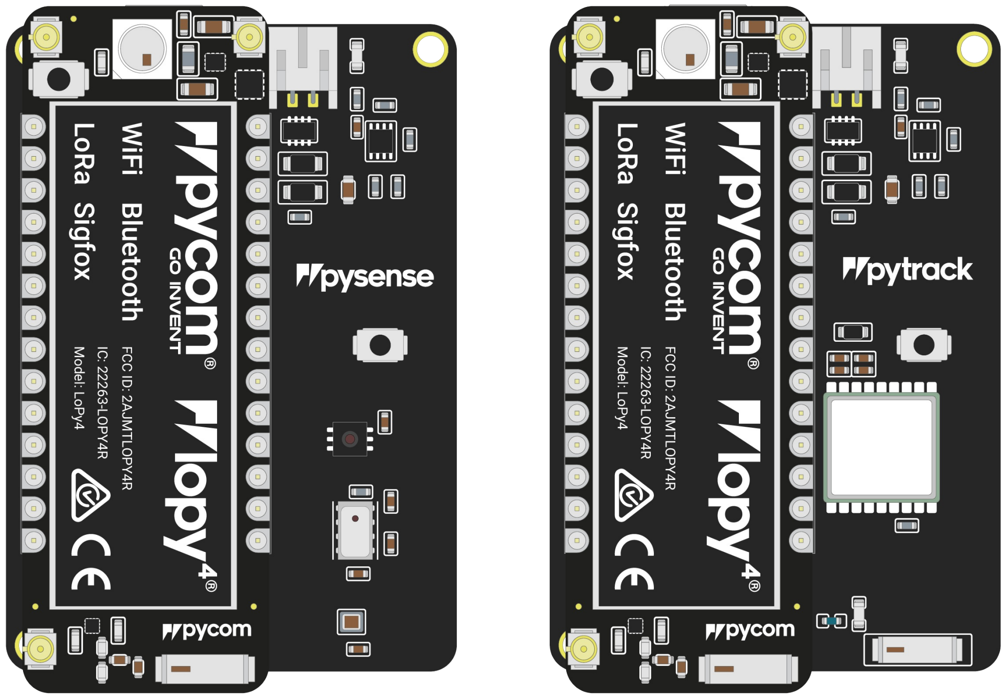
- Before connecting your module to an Expansion Board 3.0, you should update the firmware on the Expansion Board 3.0. Instructions on how to do this can be found here.

- Look for the reset button on the module (located at a corner of the board, next to the LED).

- Locate the USB connector on the expansion board.
- Insert the LoPy4 module on the Expansion Board with the reset button pointing towards the USB connector. It should firmly click into place and the pins should now no longer be visible.

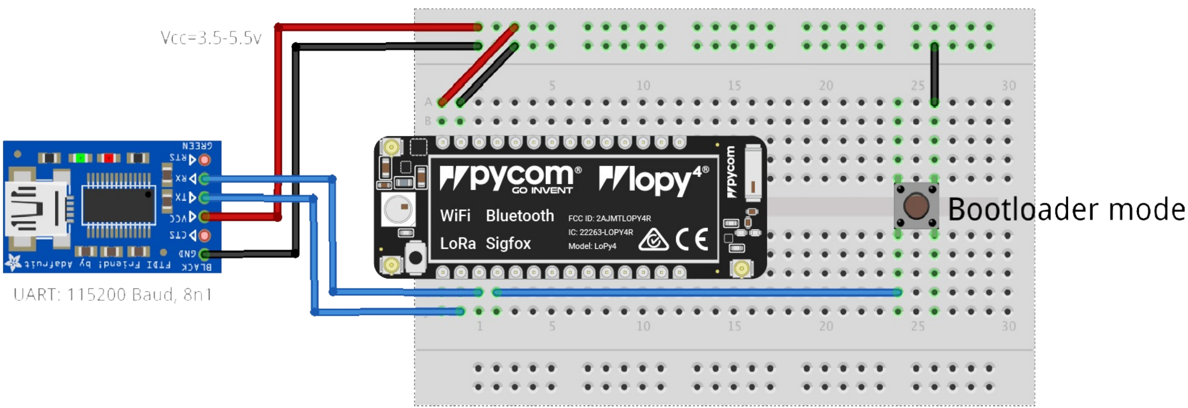


- Before connecting your module to a Pysense/Pytrack/Pyscan board, you should update the firmware on the Pysense/Pytrack/Pyscan. Instructions on how to do this can be found here.
- Look for the reset button on the LoPy4 module (located at a corner of the board, next to the LED).
- Locate the USB connector on the Pysense/Pytrack/Pyscan.
- Insert the module on the Pysense/Pytrack/Pyscan with the reset button pointing towards the USB connector. It should firmly click into place and the pins should now no longer be visible.



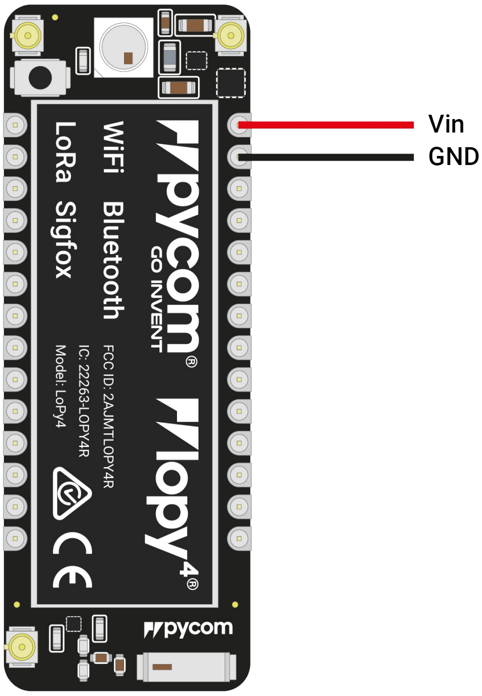
Once you have completed the above steps successfully you should see the on-board LED blinking blue. This indicates the device is powered up and running.

- Firstly you will need to connect power to your LoPy4. You will need to supply `3.5v` - `5.5v` to the `Vin` pin. **Note:** Do *not* feed `3.3v` directly to the `3.3v` supply pin, this will damage the regulator.
- The connect the `RX` and `TX` of your USB UART to the `TX` and `RX` of the LoPy4 respectively. **Note:** Please ensure you have the signal level of the UART adapter set to `3.3v` before connecting it.
- In order to put the LoPy4 into bootloader mode to update the device firmware you will need to connect `P2` to `GND`. We recommend you connect a button between the two to make this simpler.



Note: This method of connection is not recommended for first time users. It is possible to lock yourself out of the device, requiring a USB connection.

- In order to access the LoPy4 via WiFi you only need to provide `3.5v` - `5.5v` on the `Vin` pin of the LoPy4:



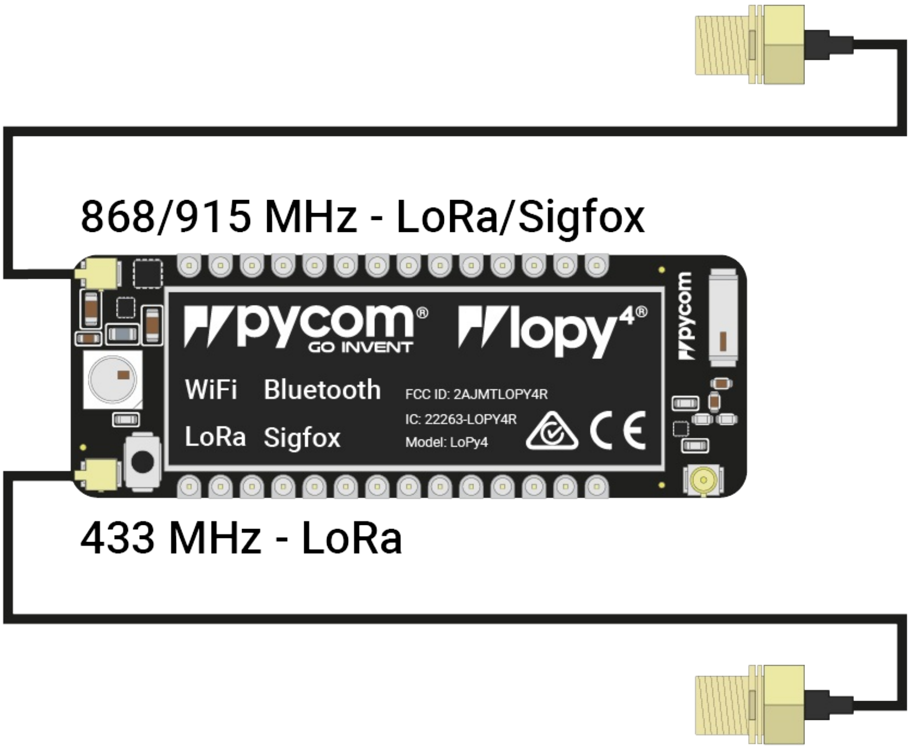
- By default, when the LoPy4 boots, it will create a WiFi access point with the following credentials:
    - SSID: `lopy4-wlan`
    - password: `www.pycom.io`
    - Once connected to this network you will be able to access the telnet and FTP servers running on the LoPy4. For both of these the login details are:
    - username: `micro`
    - password: `python`

Antennas

LoRa/Sigfox

If you intend on using the LoRa/Sigfox connectivity of the LoPy4 you **must** connect a LoRa/Sigfox antenna to your LoPy4 before trying to use LoRa/Sigfox otherwise you risk damaging the device.

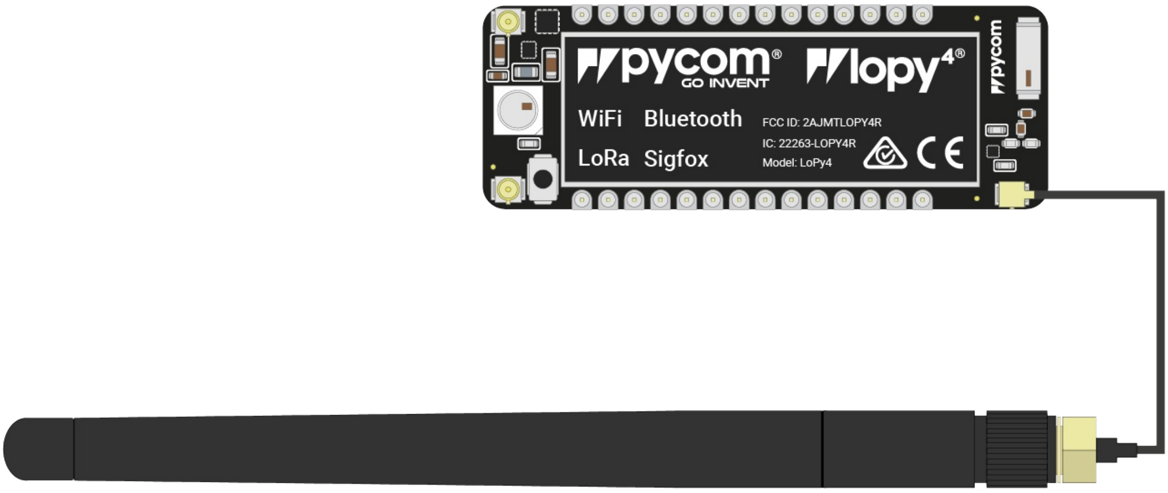
- Firstly you will need to connect the U.FL to SMA pig tail to the LoPy4 using one of the two the U.FL connectors on the same side of the LoPy4 as the LED. The one on the left hand side is for 433MHz (LoRa only), the one of the right hand side is for 868MHz/915MHz (LoRa & Sigfox). **Note:** This is different from the LoPy.



- If you are using a pycase, you will next need to put the SMA connector through the antenna hole, ensuring you align the flat edge correctly, and screw down the connector using the provided nut.
- Finally you will need to screw on the antenna to the SMA connector.

WiFi/Bluetooth (optional)

All Pycom modules, including the LoPy4, come with a on-board WiFi antenna as well as a U.FL connector for an external antenna. The external antenna is optional and only required if you need better performance or are mounting the LoPy4 in such a way that the WiFi signal is blocked. Switching between the antennas is done via software, instructions for this can be found here.

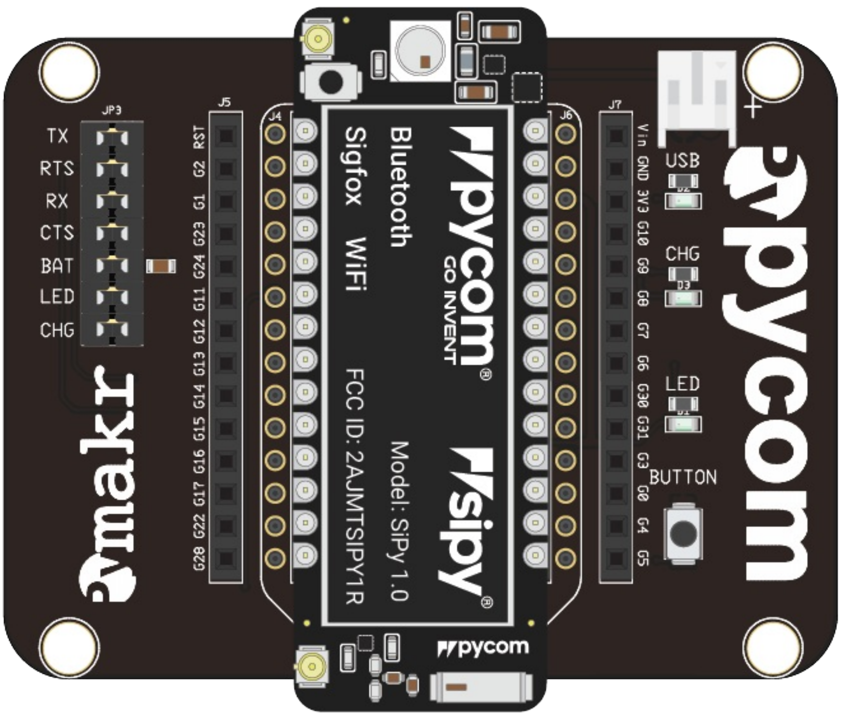


SiPy

Basic connection

- Exp Board 2.0
- Exp Board 3.0
- Pysense/Pytrack/Pyscan
- USB UART Adapter
- WiFi

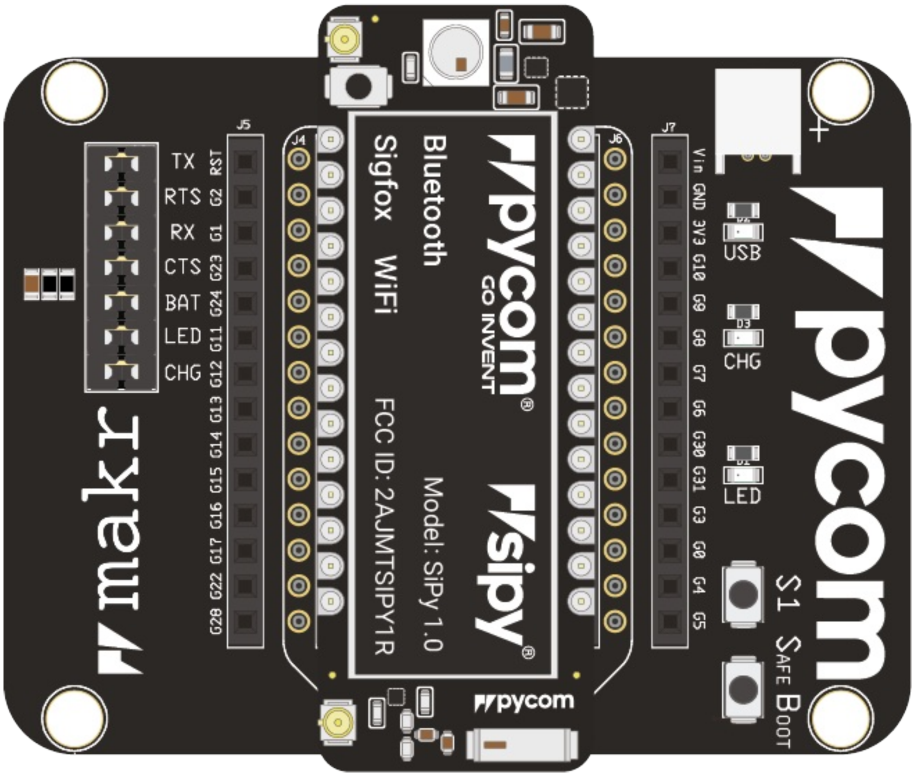
- Look for the reset button on the module (located at a corner of the board, next to the LED).
- Locate the USB connector on the expansion board.
- Insert the SiPy module on the the expansion board with the reset button pointing towards the USB connector. It should firmly click into place and the pins should now no longer be visible.



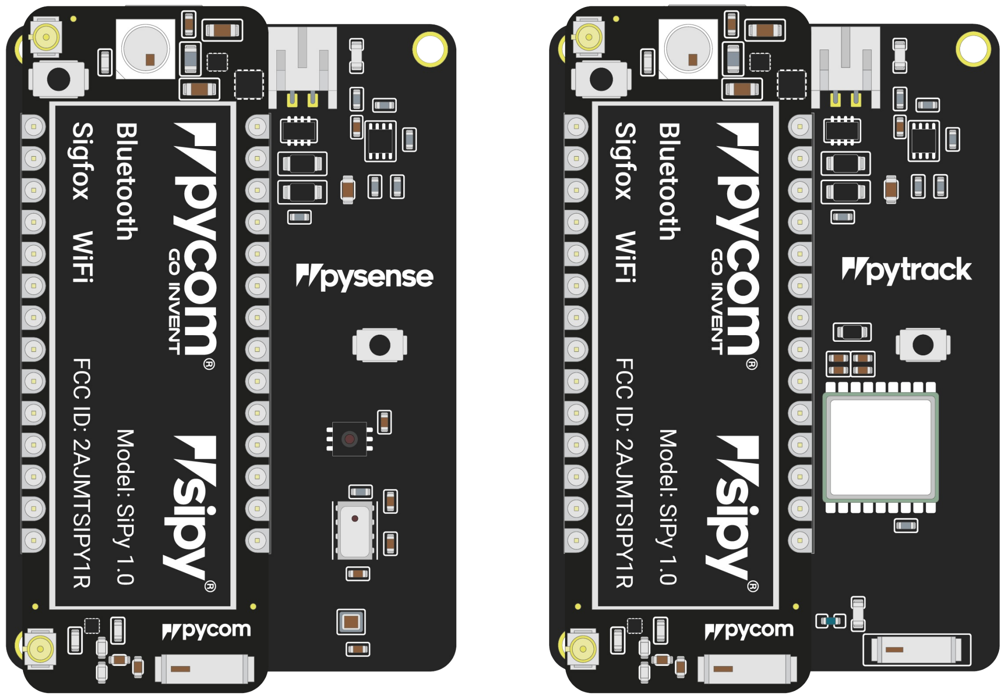
- Before connecting your module to an Expansion Board 3.0, you should update the firmware on the Expansion Board 3.0. Instructions on how to do this can be found here.

- Look for the reset button on the module (located at a corner of the board, next to the LED).

- Locate the USB connector on the expansion board.
- Insert the SiPy module on the Expansion Board with the reset button pointing towards the USB connector. It should firmly click into place and the pins should now no longer be visible.

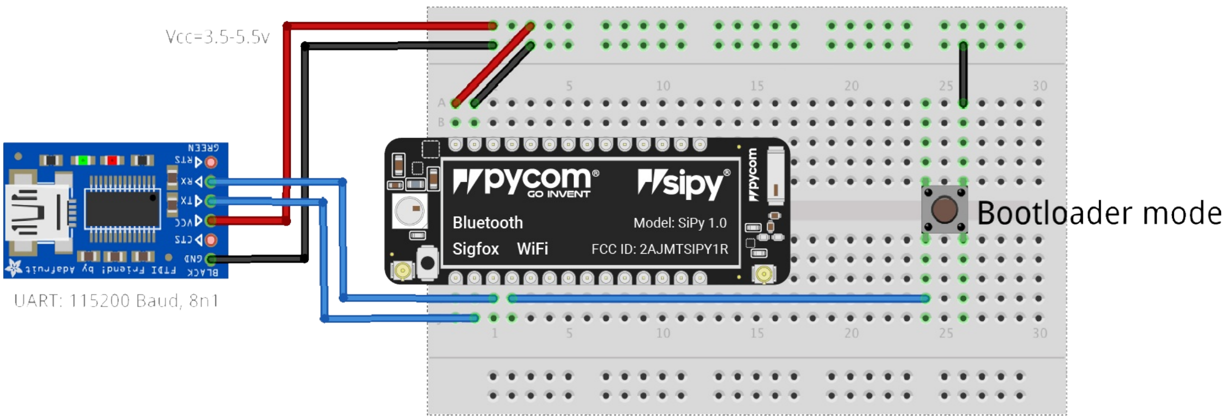


- Before connecting your module to a Pysense/Pytrack/Pyscan board, you should update the firmware on the Pysense/Pytrack/Pyscan. Instructions on how to do this can be found here.
- Look for the reset button on the SiPy module (located at a corner of the board, next to the LED).
- Locate the USB connector on the Pysense/Pytrack/Pyscan.
- Insert the module on the Pysense/Pytrack/Pyscan with the reset button pointing towards the USB connector. It should firmly click into place and the pins should now no longer be visible.



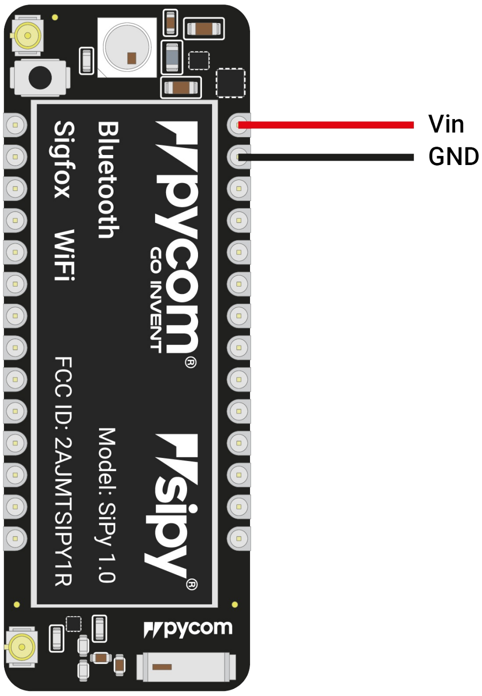
Once you have completed the above steps successfully you should see the on-board LED blinking blue. This indicates the device is powered up and running.

- Firstly you will need to connect power to your SiPy. You will need to supply `3.5v` - `5.5v` to the `Vin` pin. **Note:** Do *not* feed `3.3v` directly to the `3.3v` supply pin, this will damage the regulator.
- The connect the `RX` and `TX` of your USB UART to the `TX` and `RX` of the SiPy respectively. **Note:** Please ensure you have the signal level of the UART adapter set to `3.3v` before connecting it.
- In order to put the SiPy into bootloader mode to update the device firmware you will need to connect `P2` to `GND`. We recommend you connect a button between the two to make this simpler.



Note: This method of connection is not recommended for first time users. It is possible to lock yourself out of the device, requiring a USB connection.

- In order to access the SiPy via WiFi you only need to provide `3.5v` - `5.5v` on the `Vin` pin of the SiPy:



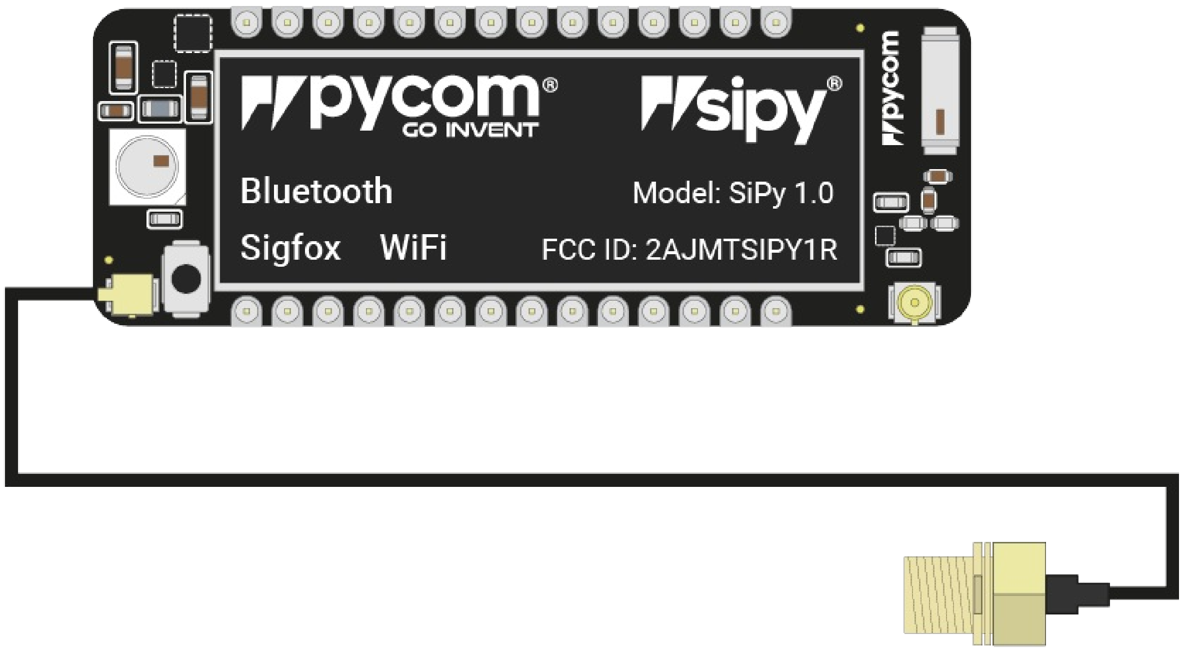
- By default, when the SiPy boots, it will create a WiFi access point with the following credentials:
    - SSID: `sipy-wlan`
    - password: `www.pycom.io`
    - Once connected to this network you will be able to access the telnet and FTP servers running on the SiPy. For both of these the login details are:
    - username: `micro`
    - password: `python`

Antennas

Sigfox

> If you intend on using the Sigfox connectivity of the SiPy you **must** connect a Sigfox antenna to your SiPy before trying to use Sigfox otherwise you risk damaging the device.

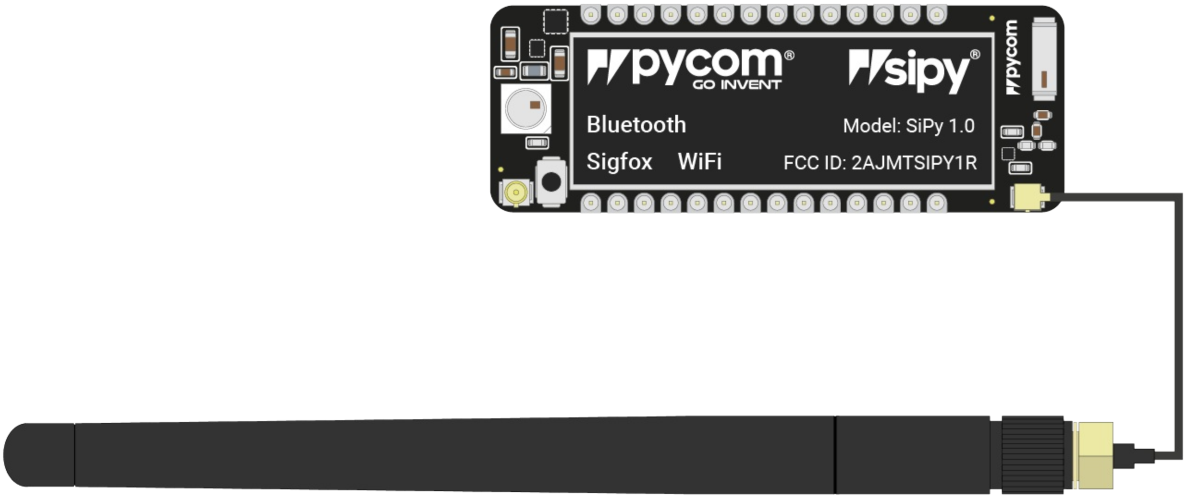
- Firstly you will need to connect the U.FL to SMA pig tail to the SiPy using the U.FL connector on the same side of the SiPy as the LED.



- If you are using a pycase, you will next need to put the SMA connector through the antenna hole, ensuring you align the flat edge correctly, and screw down the connector using the provided nut.
- Finally you will need to screw on the antenna to the SMA connector.

WiFi/Bluetooth (optional)

All Pycom modules, including the SiPy, come with a on-board WiFi antenna as well as a U.FL connector for an external antenna. The external antenna is optional and only required if you need better performance or are mounting the SiPy in such a way that the WiFi signal is blocked. Switching between the antennas is done via software, instructions for this can be found here.



Deep Sleep current issue

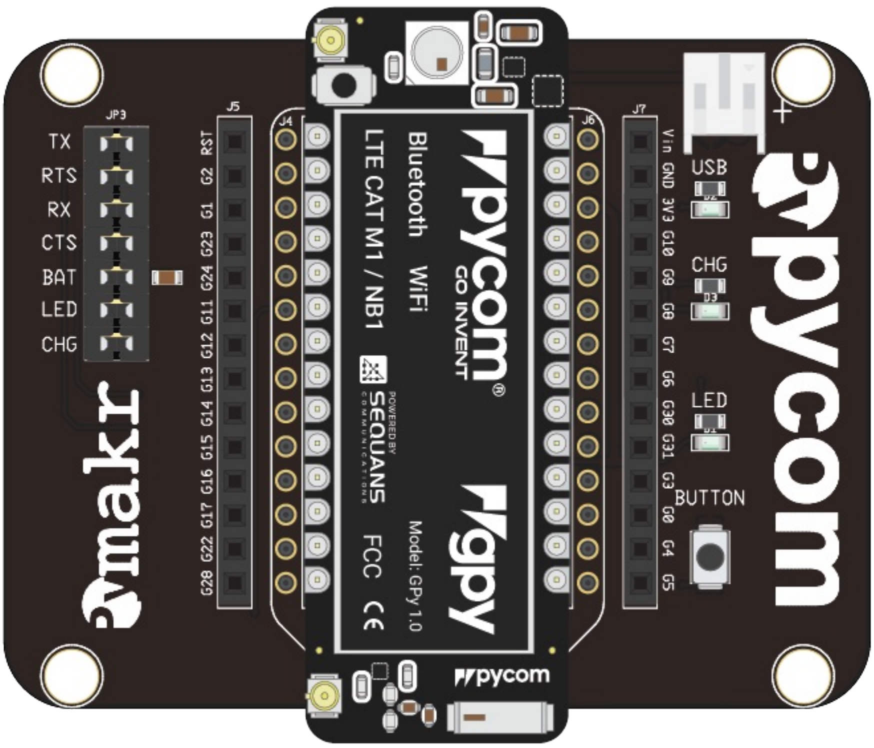
The LoPy, SiPy, and WiPy 2.0 experience an issue where the modules maintain a high current consumption in deep sleep mode. This issue has been resolved in all newer products. The cause for this issue is the DC to DC switch mode converter remains in a high performance mode even when the device is in deep sleep. The flash memory chip also does not power down. A more detailed explanation can be found here.

GPy

Basic connection

- Exp Board 2.0
- Exp Board 3.0
- Pysense/Pytrack/Pyscan
- USB UART Adapter
- WiFi

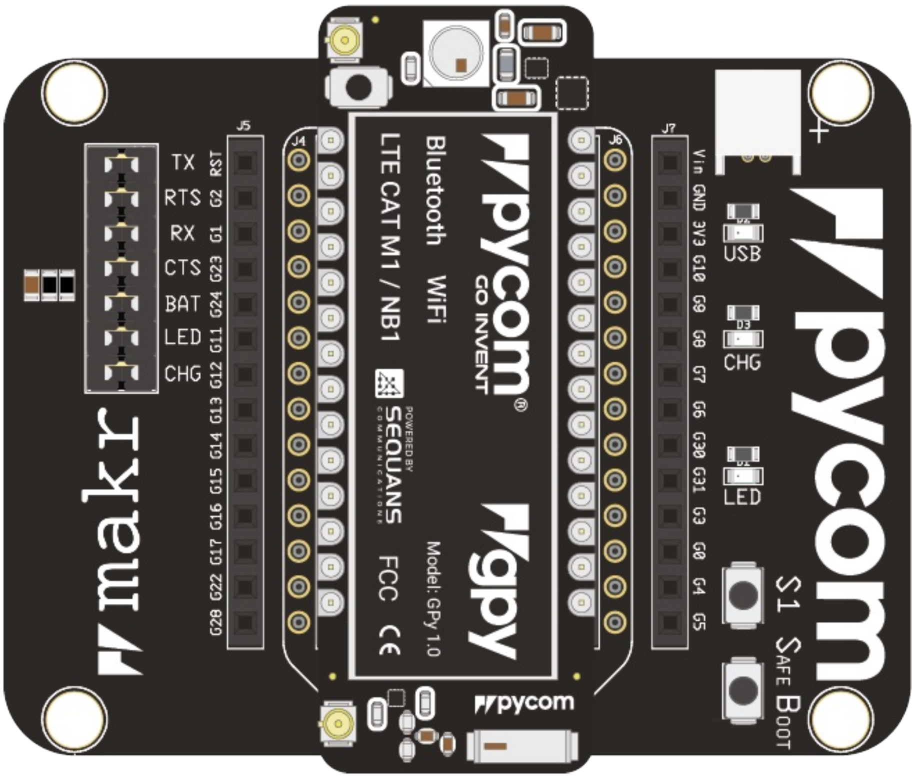
- Look for the reset button on the module (located at a corner of the board, next to the LED).
- Locate the USB connector on the expansion board.
- Insert the GPy module on the the expansion board with the reset button pointing towards the USB connector. It should firmly click into place and the pins should now no longer be visible.



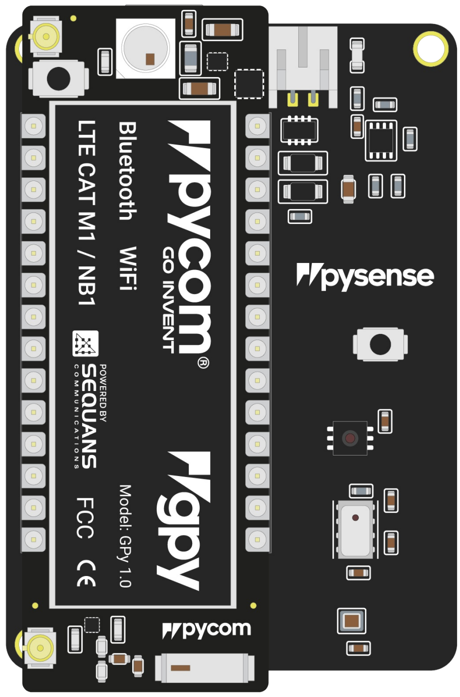
- Before connecting your module to an Expansion Board 3.0, you should update the firmware on the Expansion Board 3.0. Instructions on how to do this can be found here.

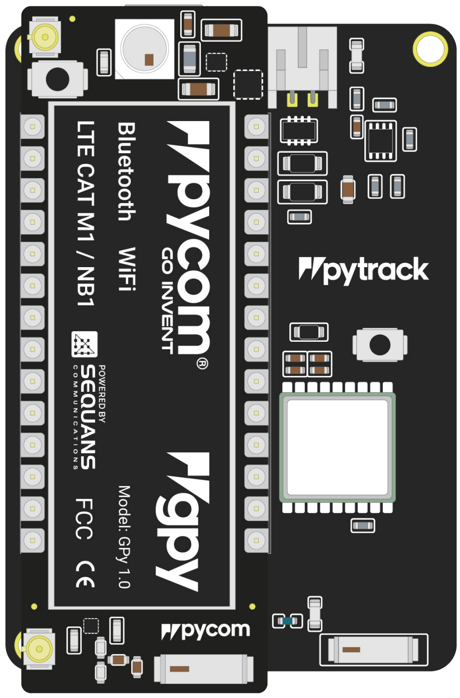
- Look for the reset button on the module (located at a corner of the board, next to the LED).

- Locate the USB connector on the expansion board.
- Insert the GPy module on the Expansion Board with the reset button pointing towards the USB connector. It should firmly click into place and the pins should now no longer be visible.



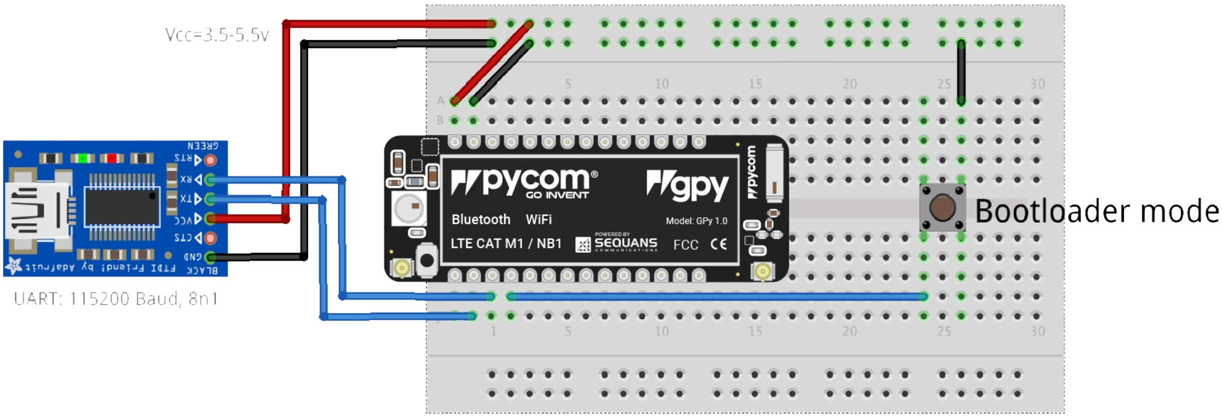
- Before connecting your module to a Pysense/Pytrack/Pyscan board, you should update the firmware on the Pysense/Pytrack/Pyscan. Instructions on how to do this can be found here.
- Look for the reset button on the GPy module (located at a corner of the board, next to the LED).
- Locate the USB connector on the Pysense/Pytrack/Pyscan.
- Insert the module on the Pysense/Pytrack/Pyscan with the reset button pointing towards the USB connector. It should firmly click into place and the pins should now no longer be visible.





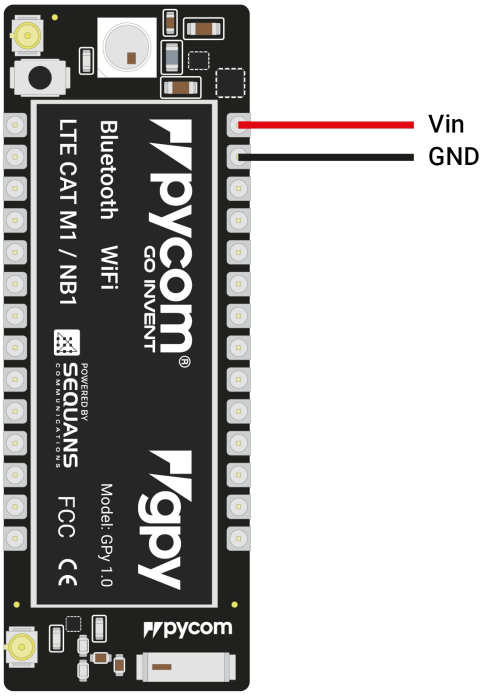
Once you have completed the above steps successfully you should see the on-board LED blinking blue. This indicates the device is powered up and running.

- Firstly you will need to connect power to your GPy. You will need to supply `3.5v` - `5.5v` to the `Vin` pin. **Note:** Do *not* feed `3.3v` directly to the `3.3v` supply pin, this will damage the regulator.
- The connect the `RX` and `TX` of your USB UART to the `TX` and `RX` of the GPy respectively. **Note:** Please ensure you have the signal level of the UART adapter set to `3.3v` before connecting it.
- In order to put the GPy into bootloader mode to update the device firmware you will need to connect `P2` to `GND`. We recommend you connect a button between the two to make this simpler.



Note: This method of connection is not recommended for first time users. It is possible to lock yourself out of the device, requiring a USB connection.

- In order to access the GPy via WiFi you only need to provide `3.5v` - `5.5v` on the `Vin` pin of the GPy:



- By default, when the GPy boots, it will create a WiFi access point with the following credentials:
    - SSID: `gpy-wlan`
    - password: `www.pycom.io`

    - Once connected to this network you will be able to access the telnet and FTP servers running on the GPy. For both of these the login details are:

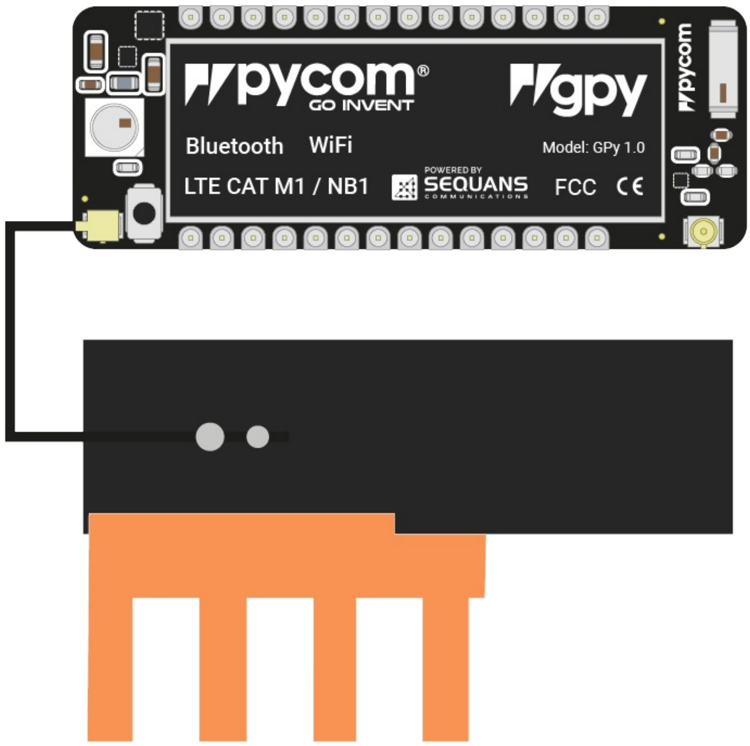
    - username: `micro`
    - password: `python`

Antennas

LTE Cat-M1/NB-IoT

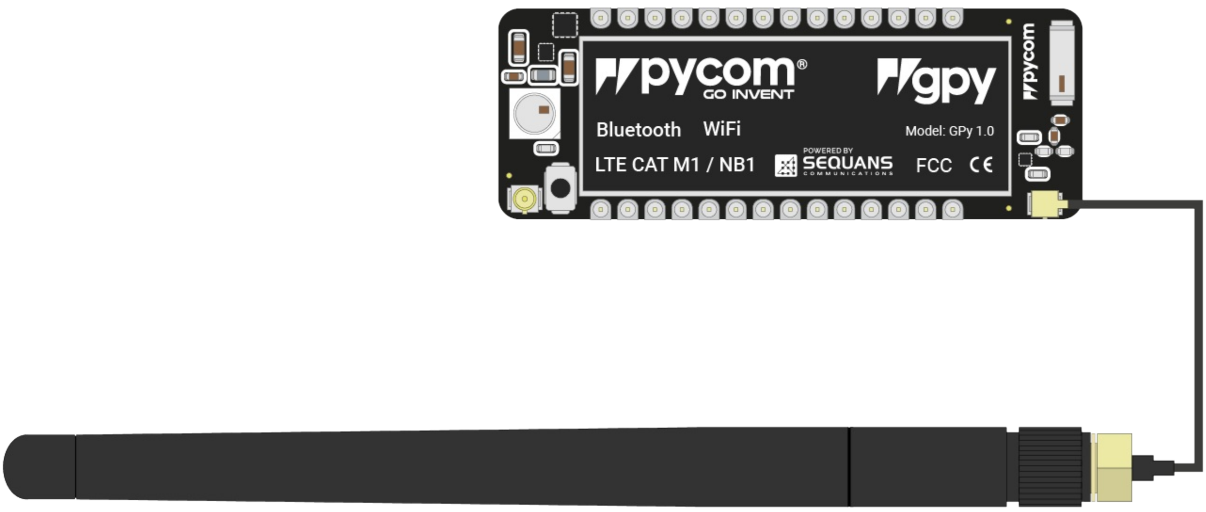
If you intend on using the LTE CAT-M1 or NB-IoT connectivity of the GPy you **must** connect a LTE CAT-M1/NB-IoT antenna to your GPy before trying to use LTE Cat-M1 or NB-IoT otherwise you risk damaging the device.

- You will need to connect the antenna to the GPy using the U.FL connector on the same side of the GPy as the LED.



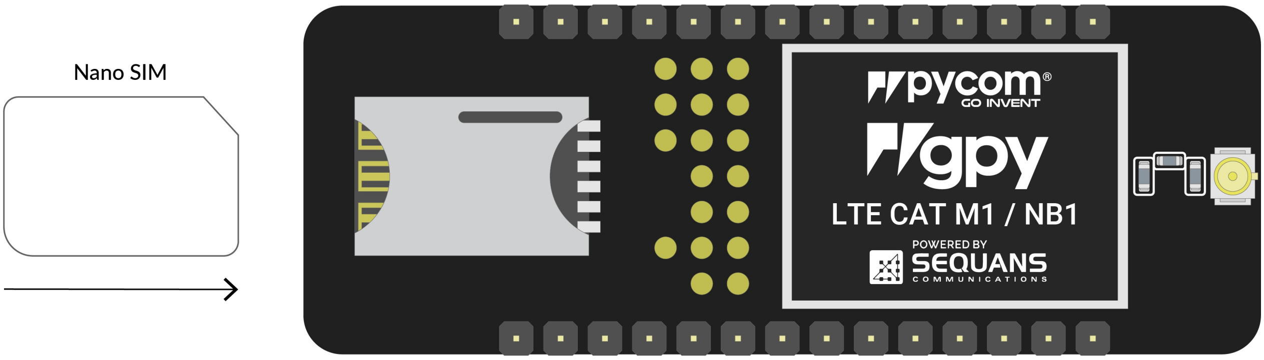
WiFi/Bluetooth (optional)

All Pycom modules, including the GPy, come with a on-board WiFi antenna as well as a U.FL connector for an external antenna. The external antenna is optional and only required if you need better performance or are mounting the GPy in such a way that the WiFi signal is blocked. Switching between the antennas is done via software, instructions for this can be found here.



SIM card

If you intend on using the LTE CAT-M1 or NB-IoT connectivity of the GPy you will need to insert a SIM card into your GPy. It should be noted that the GPy does not support regular LTE connectivity and you may require a special SIM. It is best to contact your local cellular providers for more information on acquiring a LTE CAT-M1/NB-IoT enabled nano SIM.

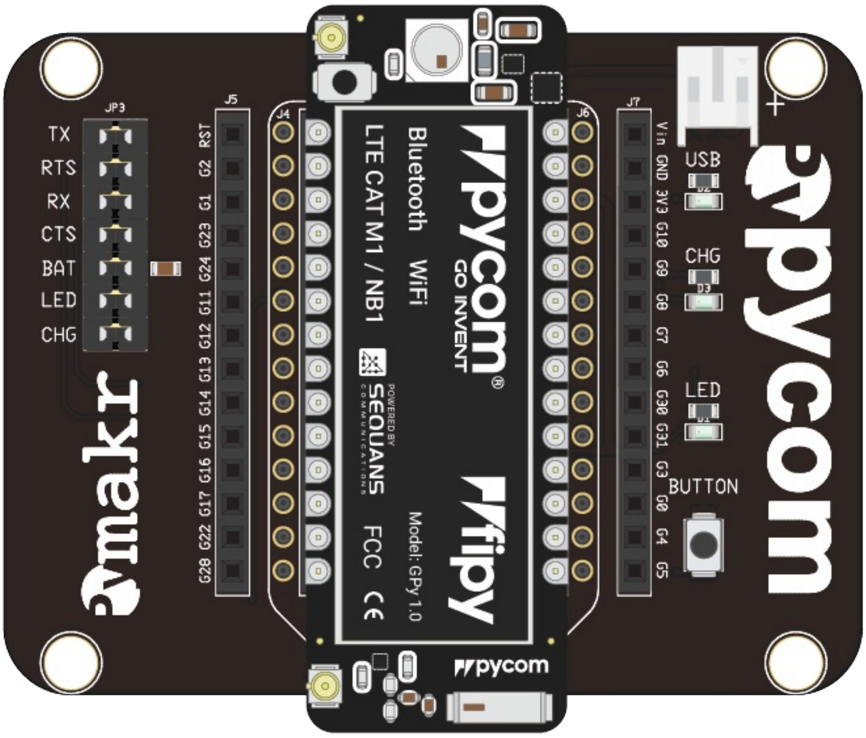


FiPy

Basic connection

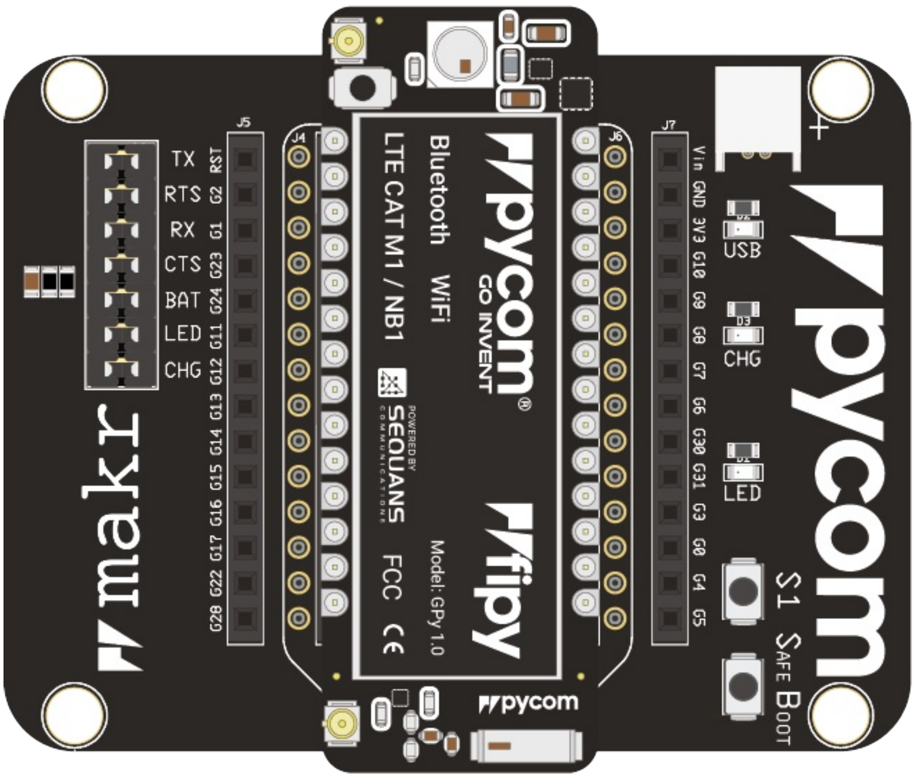
- Exp Board 2.0
- Exp Board 3.0
- Pysense/Pytrack/Pyscan
- USB UART Adapter
- WiFi

- When using the expansion board with a FiPy, you will need to remove the CTS and RTS jumpers as these interfere with communication with the cellular modem.
- Look for the reset button on the module (located at a corner of the board, next to the LED).
- Locate the USB connector on the expansion board.
- Insert the FiPy module on the the expansion board with the reset button pointing towards the USB connector. It should firmly click into place and the pins should now no longer be visible.

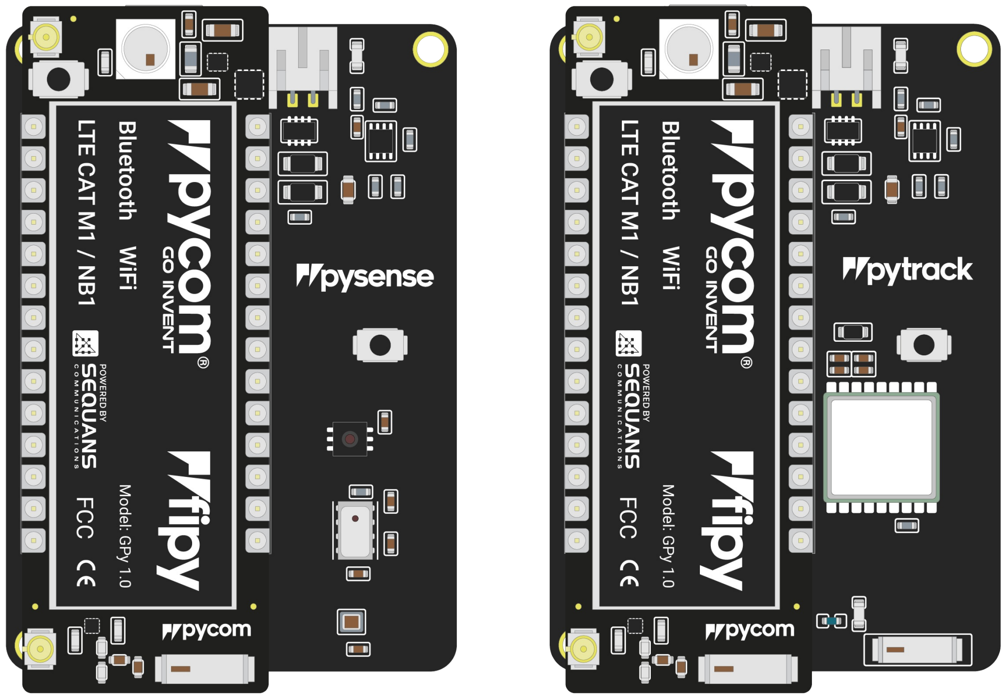


- Before connecting your module to an Expansion Board 3.0, you should update the firmware on the Expansion Board 3.0. Instructions on how to do this can be found here.

- When using the expansion board with a FiPy, you will need to remove the CTS and RTS jumpers as these interfere with communication with the cellular modem.
- Look for the reset button on the module (located at a corner of the board, next to the LED).
- Locate the USB connector on the expansion board.
- Insert the FiPy module on the Expansion Board with the reset button pointing towards the USB connector. It should firmly click into place and the pins should now no longer be visible.

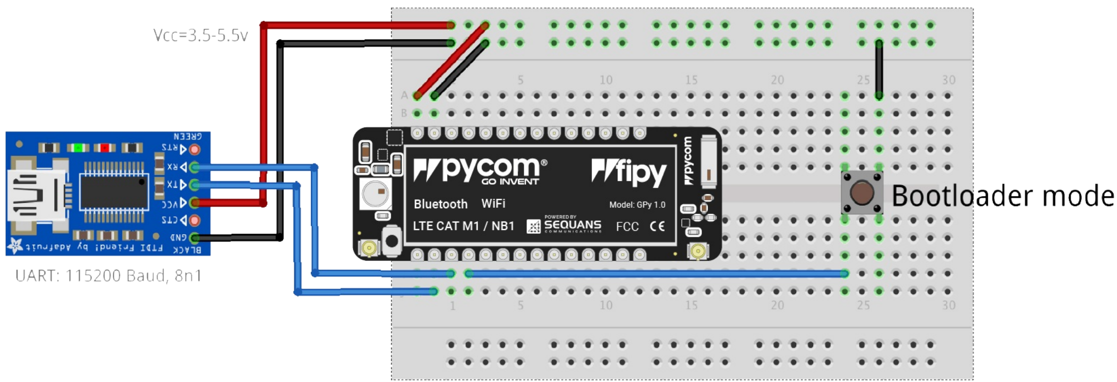


- Before connecting your module to a Pysense/Pytrack/Pyscan board, you should update the firmware on the Pysense/Pytrack/Pyscan. Instructions on how to do this can be found here.
- Look for the reset button on the FiPy module (located at a corner of the board, next to the LED).
- Locate the USB connector on the Pysense/Pytrack/Pyscan.
- Insert the module on the Pysense/Pytrack/Pyscan with the reset button pointing towards the USB connector. It should firmly click into place and the pins should now no longer be visible.



Once you have completed the above steps successfully you should see the on-board LED blinking blue. This indicates the device is powered up and running.

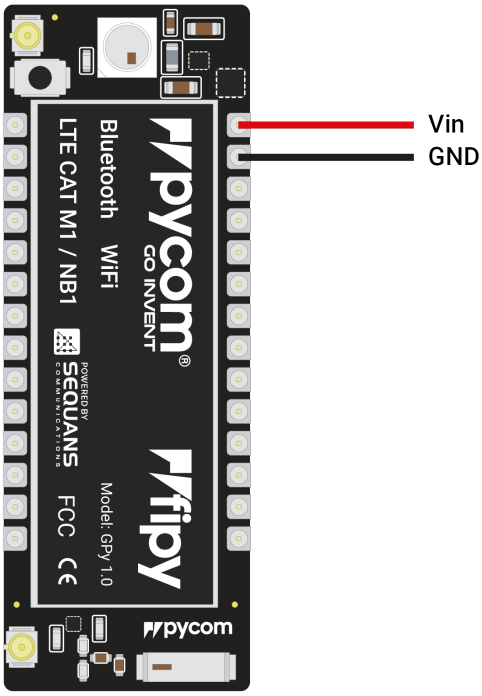
- Firstly you will need to connect power to your FiPy. You will need to supply `3.5v` - `5.5v` to the `Vin` pin. **Note:** Do *not* feed `3.3v` directly to the `3.3v` supply pin, this will damage the regulator.
- The connect the `RX` and `TX` of your USB UART to the `TX` and `RX` of the FiPy respectively. **Note:** Please ensure you have the signal level of the UART adapter set to `3.3v` before connecting it.
- In order to put the FiPy into bootloader mode to update the device firmware you will need to connect `P2` to `GND`. We recommend you connect a button between the two to make this simpler.



Note: This method of connection is not recommended for first time users. It is possible to lock yourself out of the device, requiring a USB connection.

- In order to access the FiPy via WiFi you only need to provide `3.5v` - `5.5v` on the

`Vin` pin of the FiPy:



- By default, when the FiPy boots, it will create a WiFi access point with the following credentials:
  - SSID: `fipy-wlan`
  - password: `www.pycom.io`

  - Once connected to this network you will be able to access the telnet and FTP servers running on the FiPy. For both of these the login details are:

  - username: `micro`
  - password: `python`

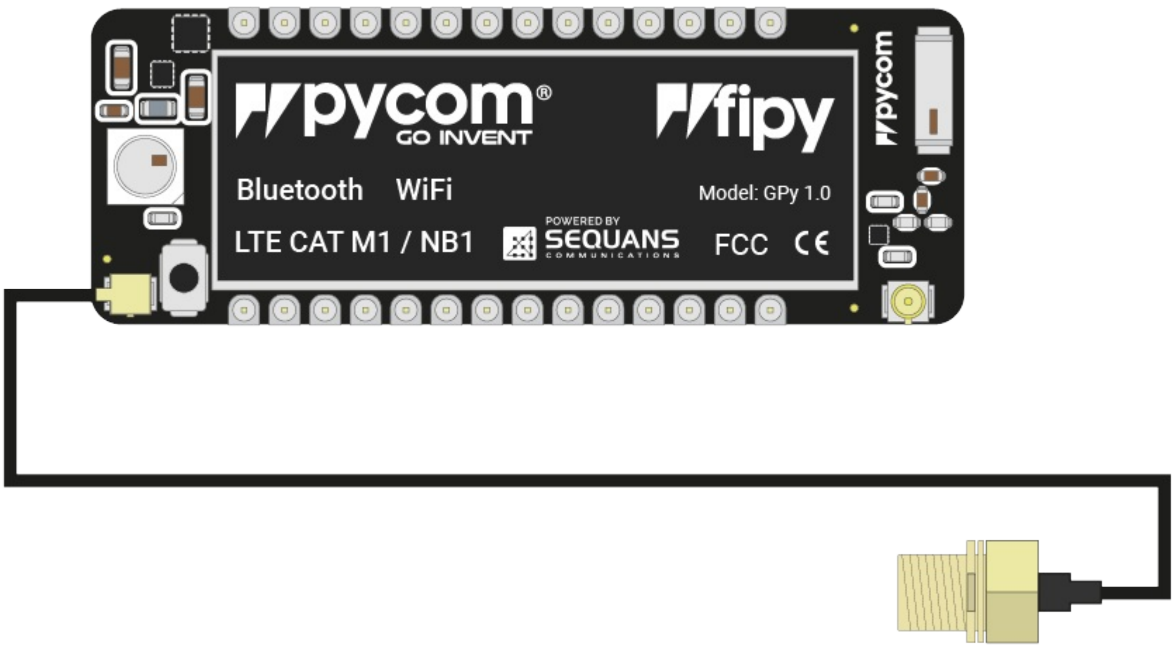
Antennas

LoRa/Sigfox

If you intend on using the LoRa/Sigfox connectivity of the FiPy you **must** connect a LoRa/Sigfox antenna to your FiPy before trying to use LoRa/Sigfox otherwise you risk damaging the device.

The FiPy only supports LoRa on the 868MHz or 915MHz bands. It does not support 433MHz. For this you will require a LoPy4.

- Firstly you will need to connect the U.FL to SMA pig tail to the FiPy using the U.FL connector on the same side of the FiPy as the LED.

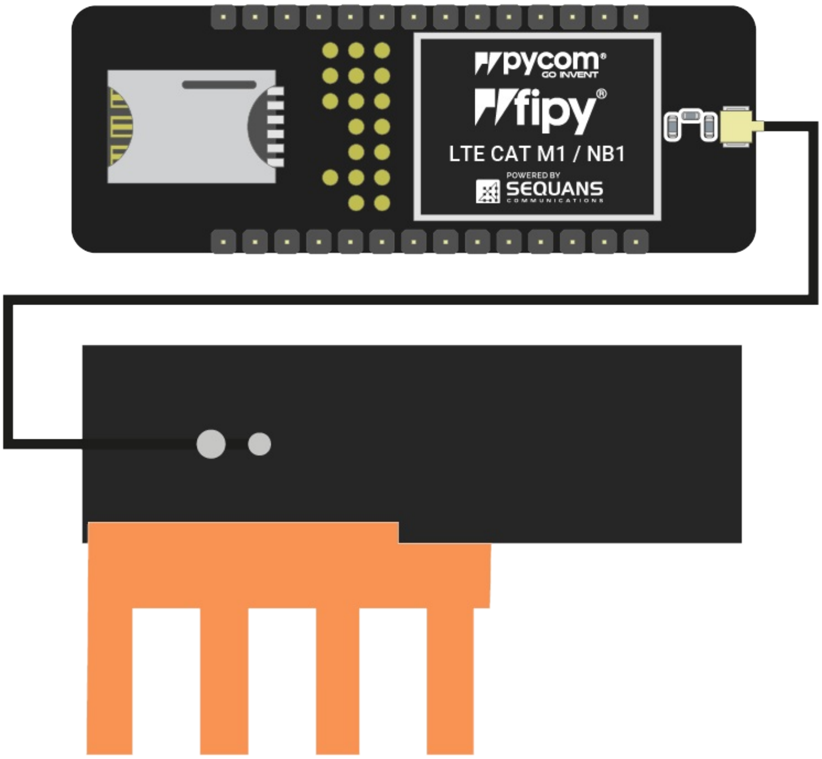


- If you are using a pycase, you will next need to put the SMA connector through the antenna hole, ensuring you align the flat edge correctly, and screw down the connector using the provided nut.
- Finally you will need to screw on the antenna to the SMA connector.

LTE Cat-M1/NB-IoT

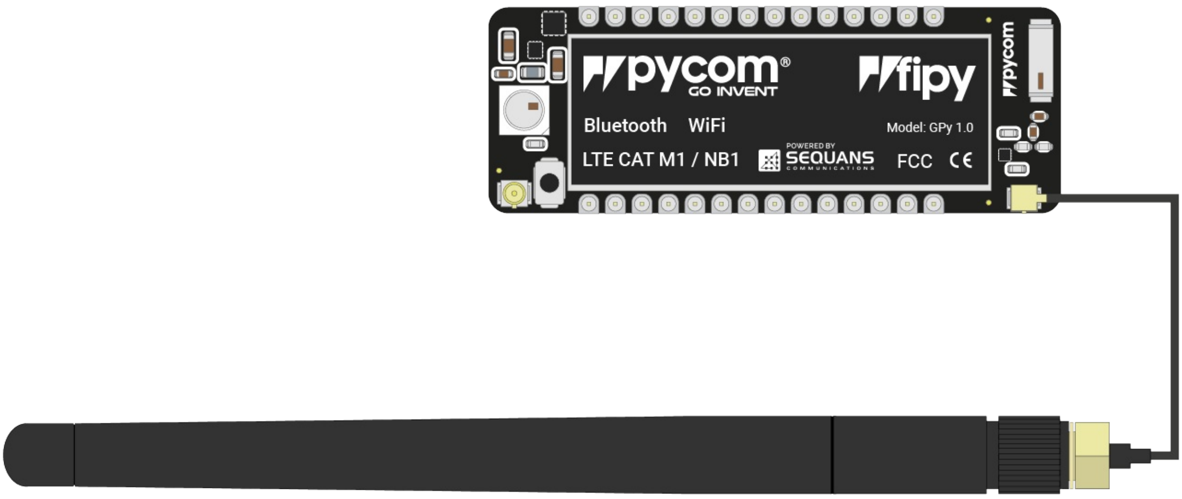
If you intend on using the LTE CAT-M1 or NB-IoT connectivity of the FiPy you **must** connect a LTE CAT-M1/NB-IoT antenna to your FiPy before trying to use LTE Cat-M1 or NB-IoT otherwise you risk damaging the device.

- You will need to connect the antenna to the FiPy using the U.FL connector on the under side of the FiPy.



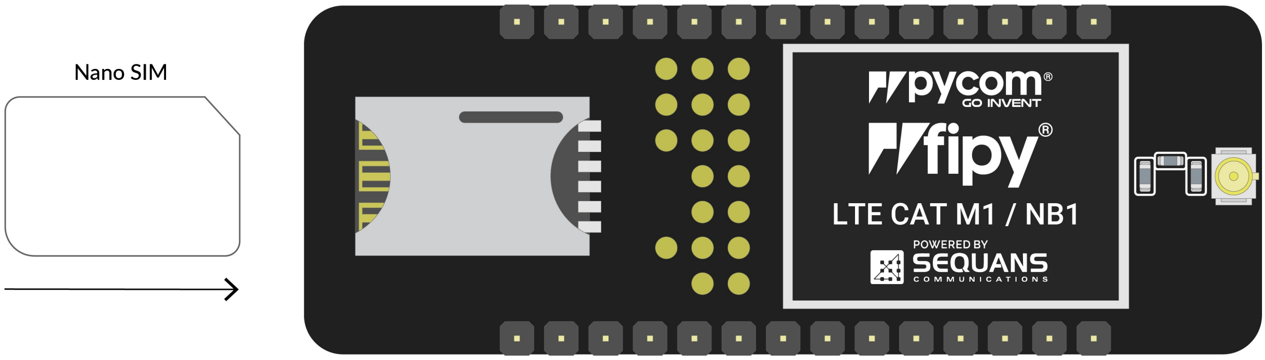
WiFi/Bluetooth (optional)

All Pycom modules, including the FiPy, come with a on-board WiFi antenna as well as a U.FL connector for an external antenna. The external antenna is optional and only required if you need better performance or are mounting the FiPy in such a way that the WiFi signal is blocked. Switching between the antennas is done via software, instructions for this can be found here.



SIM card

If you intend on using the LTE CAT-M1 or NB-IoT connectivity of the FiPy you will need to insert a SIM card into your FiPy. It should be noted that the FiPy does not support regular LTE connectivity and you may require a special SIM. It is best to contact your local cellular providers for more information on acquiring a LTE CAT-M1/NB-IoT enabled nano SIM.

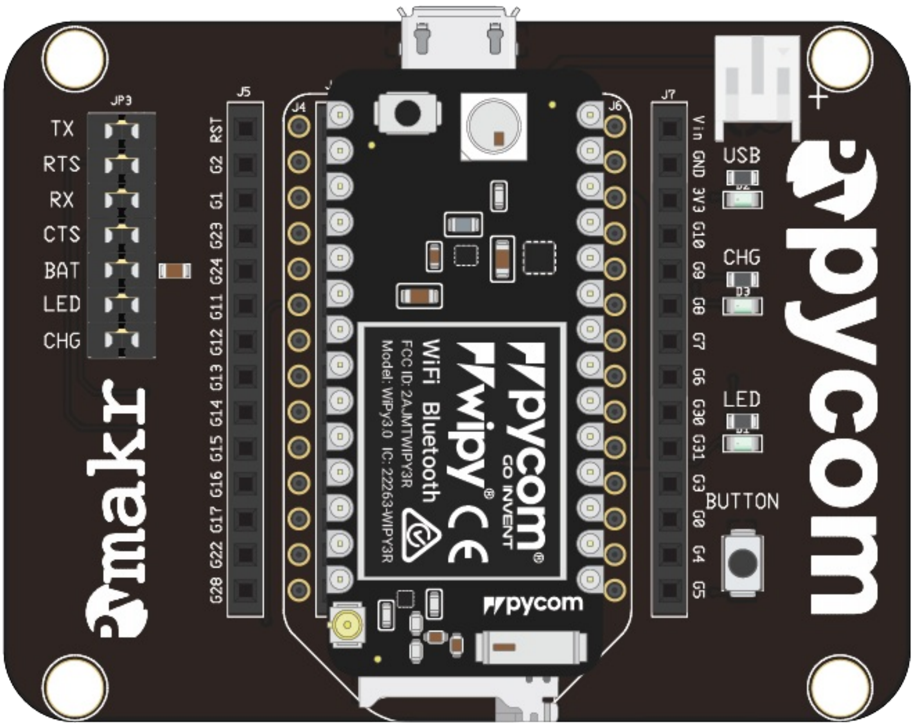


WiPy

Basic connection

- Exp Board 2.0
- Exp Board 3.0
- Pysense/Pytrack/Pyscan
- USB UART Adapter
- WiFi

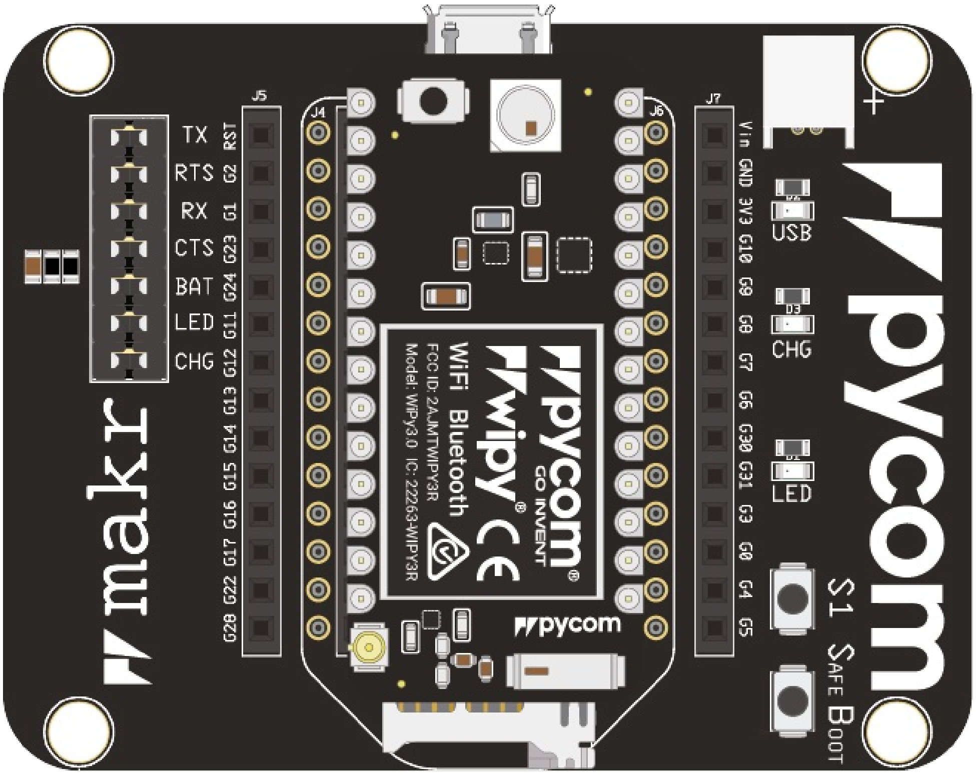
- Look for the reset button on the module (located at a corner of the board, next to the LED).
- Locate the USB connector on the expansion board.
- Insert the WiPy module on the the expansion board with the reset button pointing towards the USB connector. It should firmly click into place and the pins should now no longer be visible.



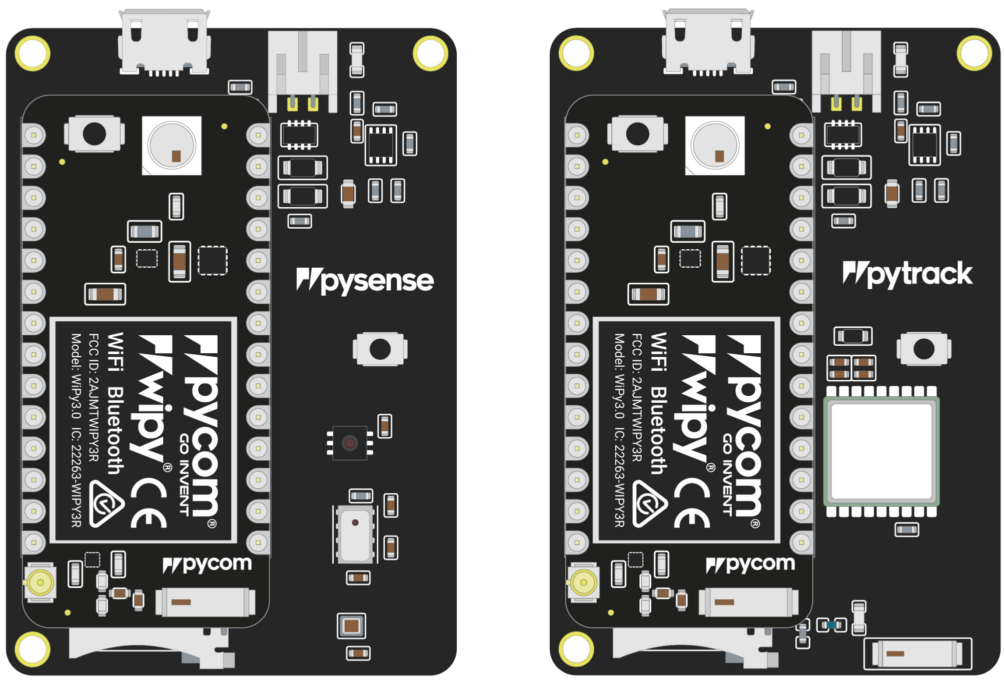
- Before connecting your module to an Expansion Board 3.0, you should update the firmware on the Expansion Board 3.0. Instructions on how to do this can be found here.

- Look for the reset button on the module (located at a corner of the board, next to the LED).

- Locate the USB connector on the expansion board.
- Insert the WiPy module on the Expansion Board with the reset button pointing towards the USB connector. It should firmly click into place and the pins should now no longer be visible.

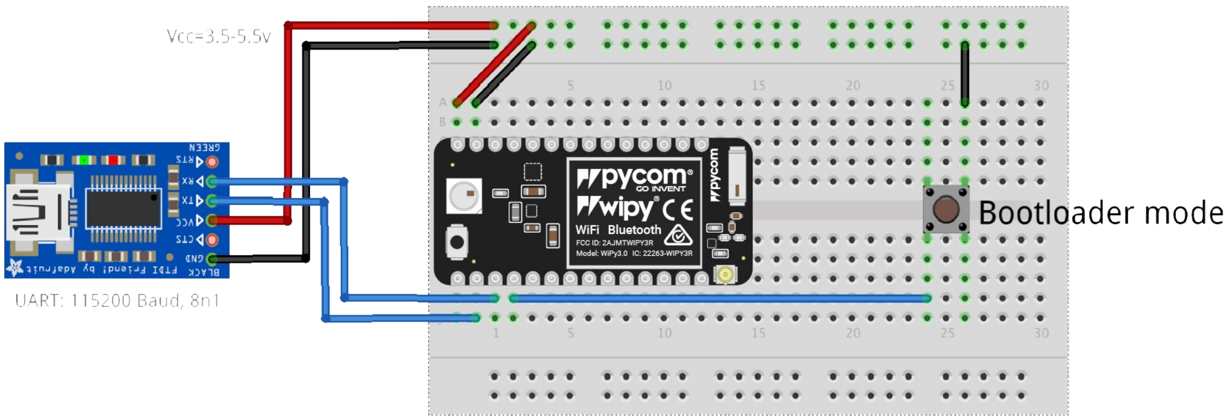


- Before connecting your module to a Pysense/Pytrack/Pyscan board, you should update the firmware on the Pysense/Pytrack/Pyscan. Instructions on how to do this can be found here.
- Look for the reset button on the WiPy module (located at a corner of the board, next to the LED).
- Locate the USB connector on the Pysense/Pytrack/Pyscan.
- Insert the module on the Pysense/Pytrack/Pyscan with the reset button pointing towards the USB connector. It should firmly click into place and the pins should now no longer be visible.



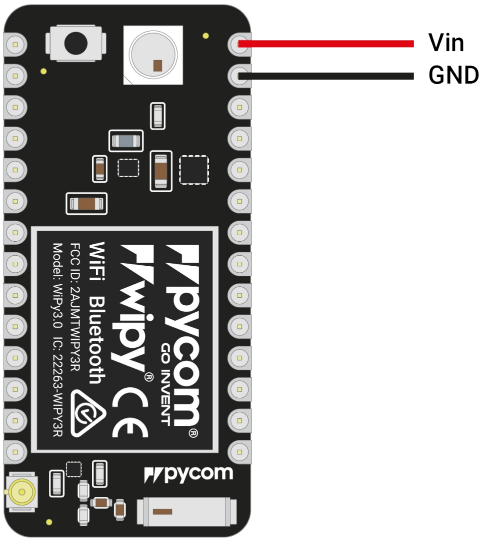
Once you have completed the above steps successfully you should see the on-board LED blinking blue. This indicates the device is powered up and running.

- Firstly you will need to connect power to your WiPy. You will need to supply `3.5v` - `5.5v` to the `Vin` pin. **Note:** Do *not* feed `3.3v` directly to the `3.3v` supply pin, this will damage the regulator.
- The connect the `RX` and `TX` of your USB UART to the `TX` and `RX` of the WiPy respectively. **Note:** Please ensure you have the signal level of the UART adapter set to `3.3v` before connecting it.
- In order to put the WiPy into bootloader mode to update the device firmware you will need to connect `P2` to `GND`. We recommend you connect a button between the two to make this simpler.



Note: This method of connection is not recommended for first time users. It is possible to lock yourself out of the device, requiring a USB connection.

- In order to access the WiPy via WiFi you only need to provide `3.5v` - `5.5v` on the `Vin` pin of the WiPy:

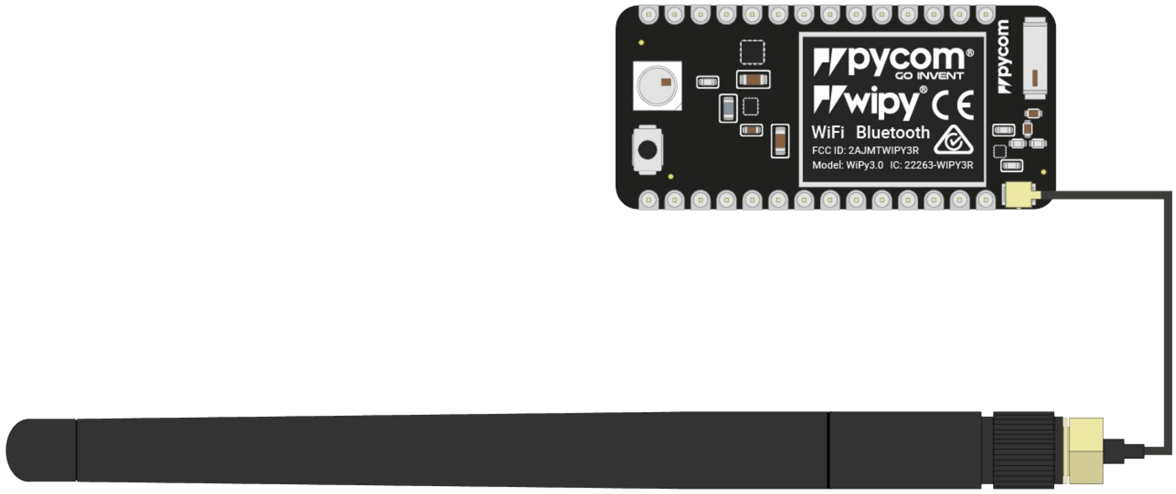


- By default, when the WiPy boots, it will create a WiFi access point with the following credentials:
  - SSID: `wipy-wlan`
  - password: `www.pycom.io`
  - Once connected to this network you will be able to access the telnet and FTP servers running on the WiPy. For both of these the login details are:
  - username: `micro`
  - password: `python`

Antennas

WiFi/Bluetooth (optional)

All Pycom modules, including the WiPy, come with a on-board WiFi antenna as well as a U.FL connector for an external antenna. The external antenna is optional and only required if you need better performance or are mounting the WiPy in such a way that the WiFi signal is blocked. Switching between the antennas is done via software, instructions for this can be found here.



Deep Sleep current issue

The LoPy, SiPy, and WiPy 2.0 experience an issue where the modules maintain a high current consumption in deep sleep mode. This issue has been resolved in all newer products. The cause for this issue is the DC to DC switch mode converter remains in a high performance mode even when the device is in deep sleep. The flash memory chip also does not power down. A more detailed explanation can be found here.

WiPy 2.0 vs WiPy 3.0

The WiPy 3.0 is an upgraded version of the WiPy 2.0 with the following changes:

- The FLASH has been upgraded from 4MB to 8MB.
- The RAM has been upgraded from 512KB to 4MB.
- The deepsleep current consumption issue has been fixed
- The antenna select pin has moved to GPIO21 (P12)

Setting up your computer

To get you up and running, Pycom provides a suite of tools to assist with developing and programming your Pycom Devices:

1. **Drivers:** If you are using Microsoft Windows, you might be required to install drivers for our products to function correctly.
2. **Pycom firmware update utility:** This tool automates the process of upgrading the firmware of your Pycom device. It is important that you use this tool before you attempt to use your device. Not only to ensure you have the most stable and feature packed firmware, but also to ensure all the functionality of your device is enable. E.g. this tool also activates your two year free sigfox connectivity.
3. **Development Environment:** Pymakr is a plug-in for Atom and Visual Studio Code developed by Pycom to make development for Pycom modules super easy. It allows you to use your favourite text editor while simplifying the process of uploading code to the device.

Drivers

Linux

You should not need to install any drivers for our devices to be recognised by Linux. You may how ever need to adjust permissions to make sure you have access to the serial port. On most distributions this can be done by adding your user to the `dialout` user group. Please check the specific instructions for your linux distribution for how to do this.

macOS

On macOS you shouldn't need to do anything special to get our device to work.

Windows

All our products will work out of the box for Windows 8/10/+. If using Windows 7, drivers to support the Pysense/Pytrack/Pyscan/Expansion Board 3.0 boards will need to be installed.

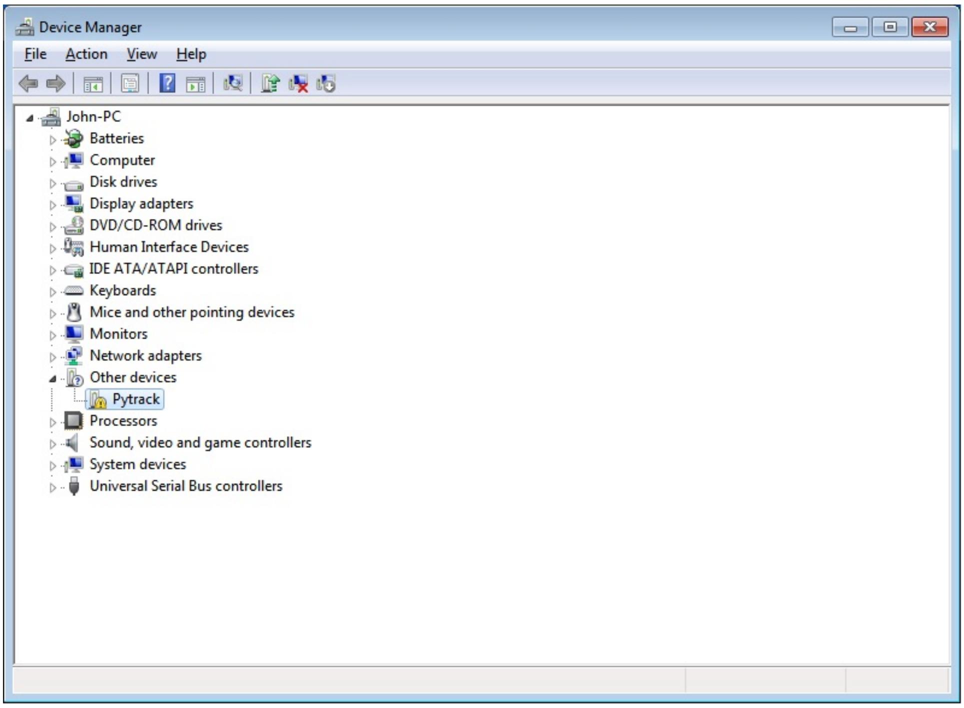
Download

Please download the driver software from the link below.

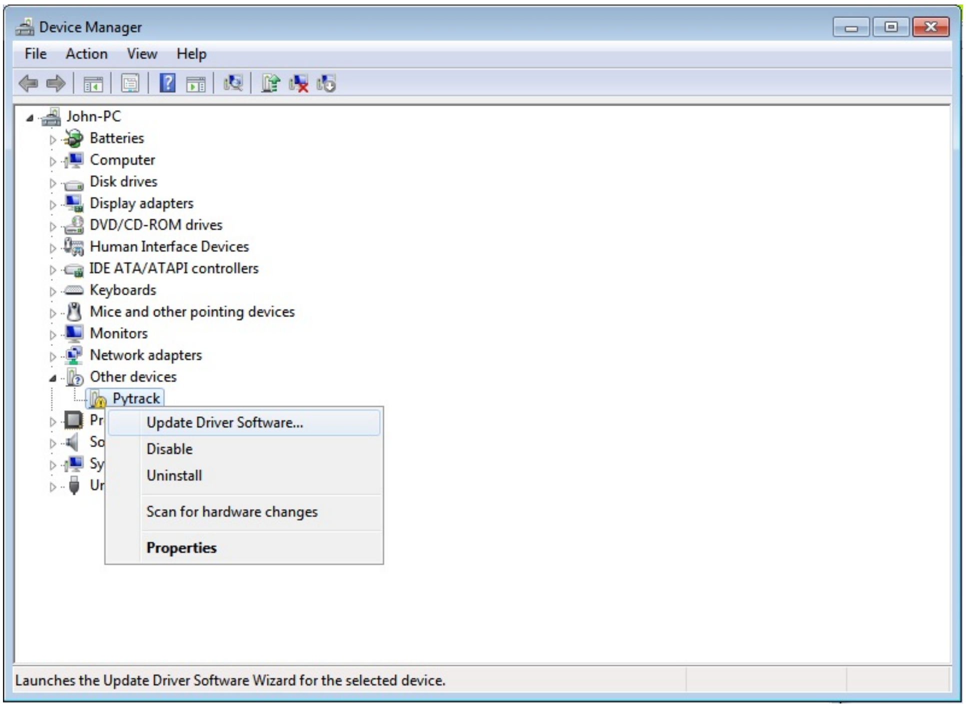
Pysense/Pytrack/Pyscan/Expansion Board 3.0 Serial Driver

Installation

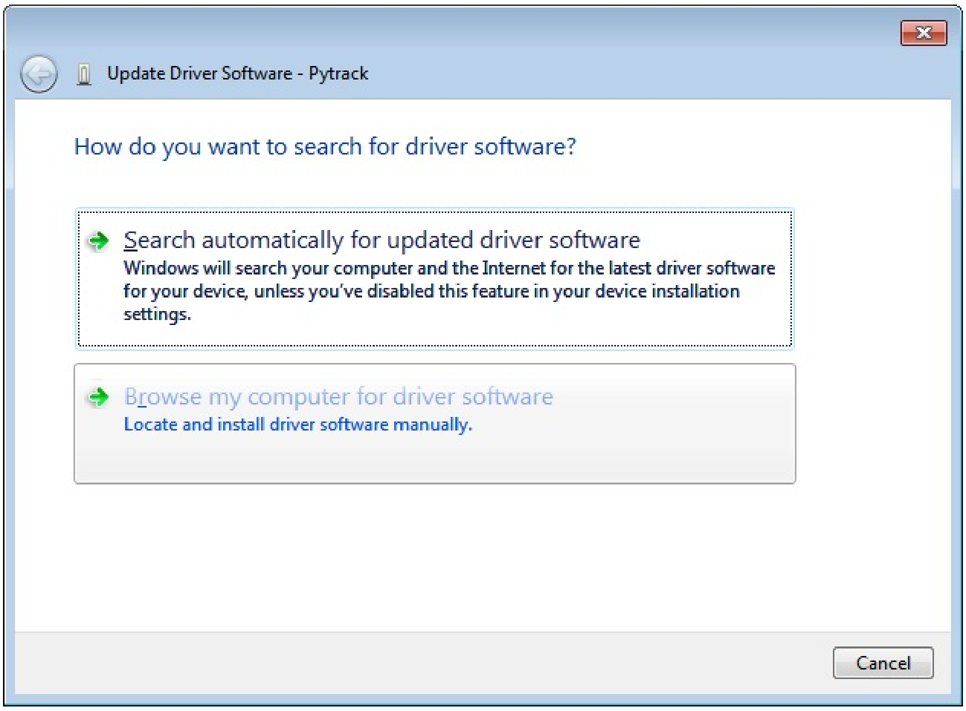
First navigate open the Windows start menu and search/navigate to `Device Manager. You should see your Pytrack/Pysense in the dropdown under **other devices**.



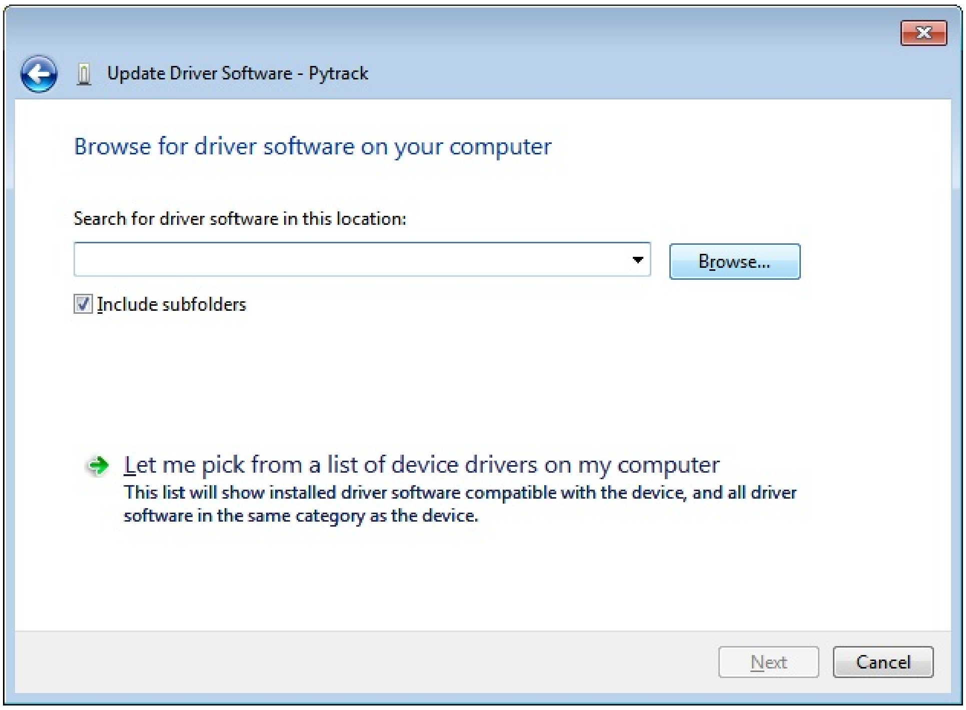
Right click the device and select `Update Driver Software` .



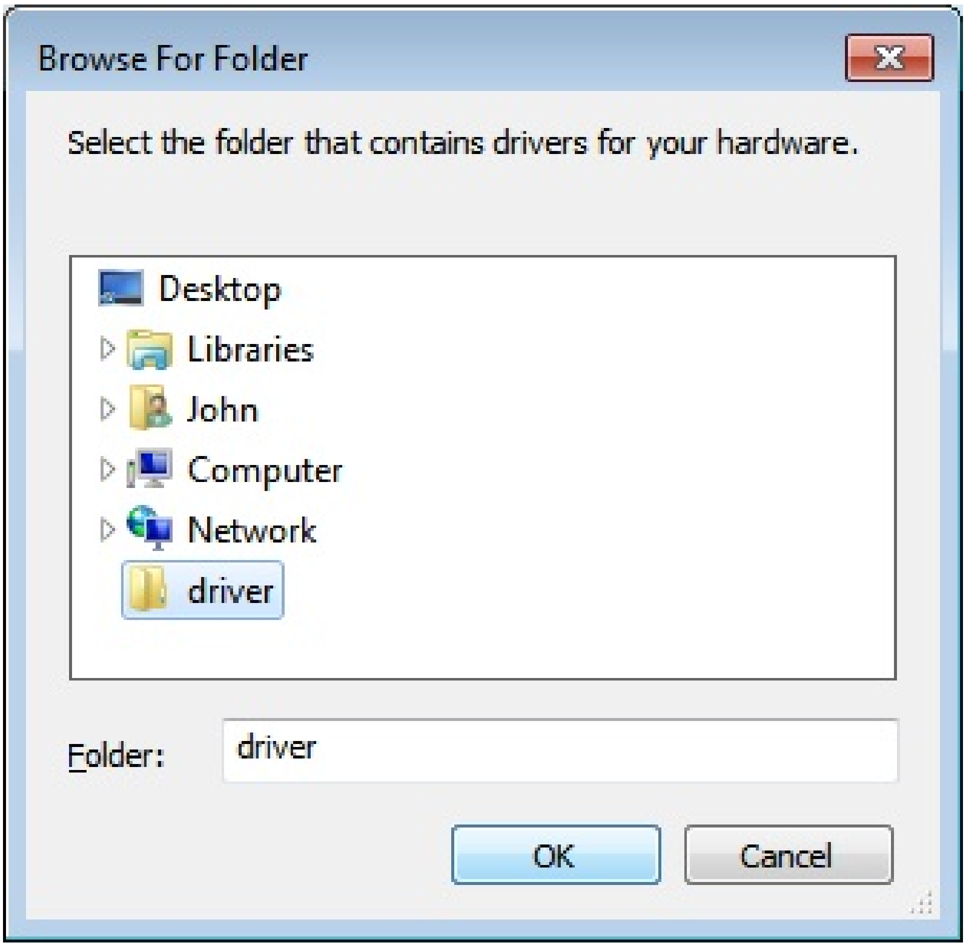
Select the option to **Browse my computer for driver software**.



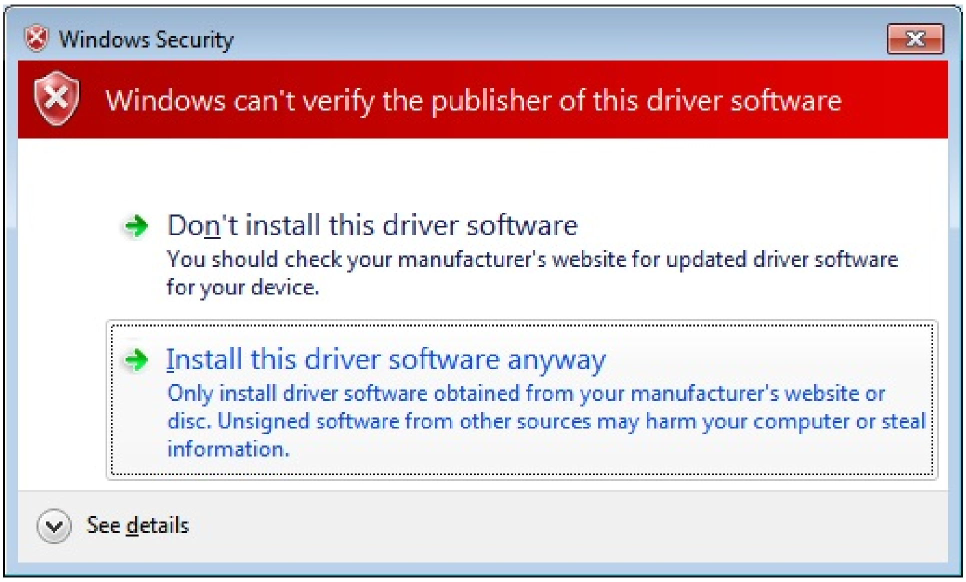
Next you will need to navigate to where you downloaded the driver to (e.g. **Downloads** Folder).



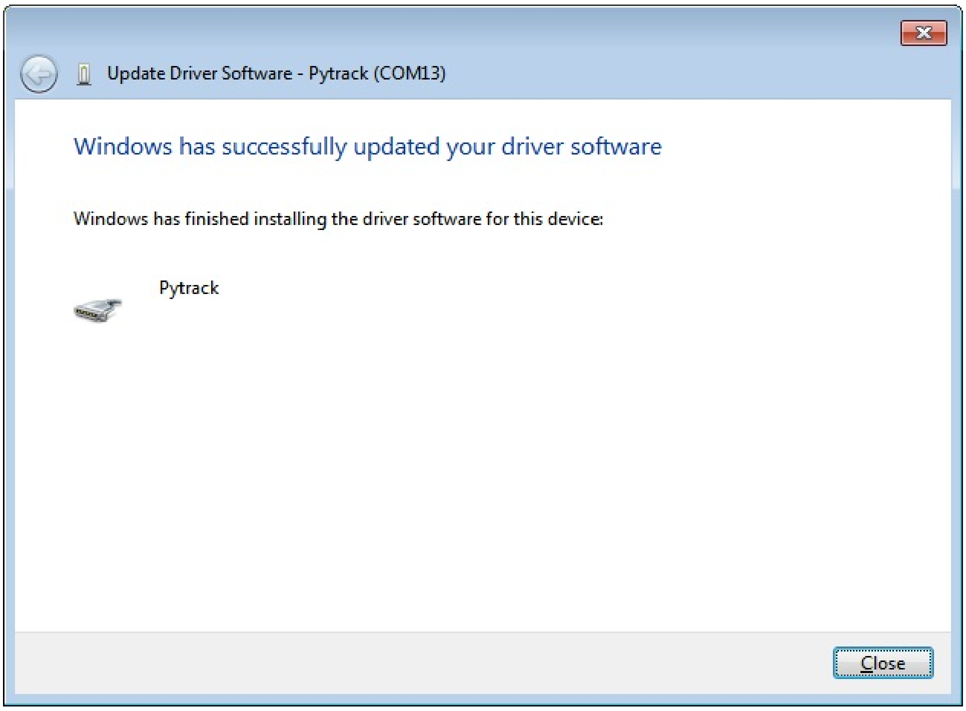
Specify the folder in which the drivers are contained. If you haven't extracted the `.zip` file, please do this before selecting the folder.



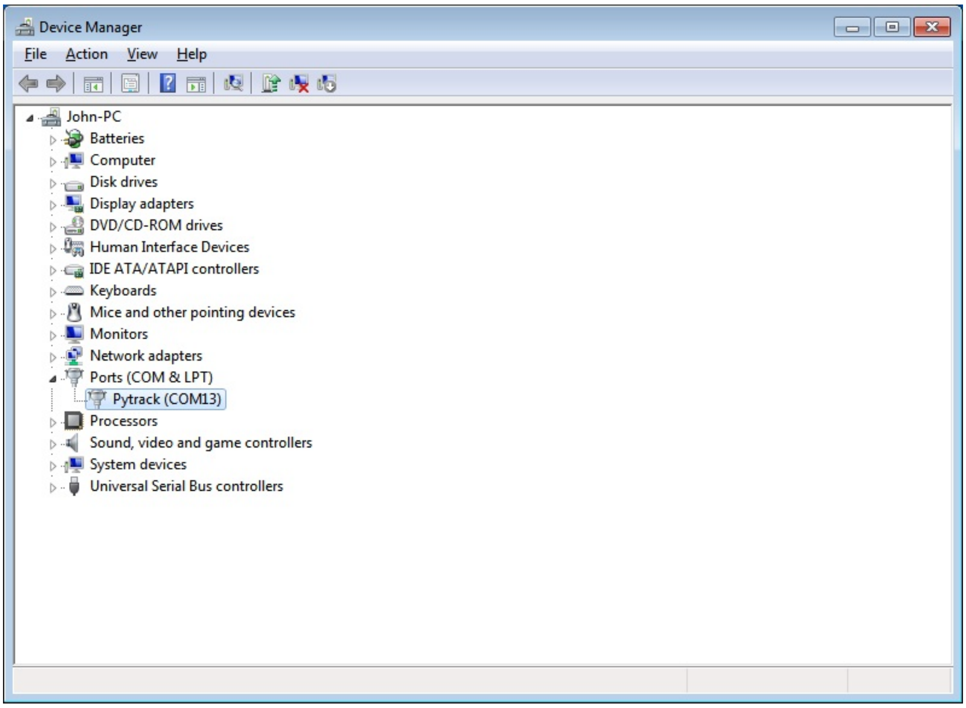
You may receive a warning, suggesting that Windows can't verify the publisher of this driver. Click `Install this driver software anyway` as this link points to our official driver.



If the installation was successful, you should now see a window specifying that the driver was correctly installed.



To confirm that the installation was correct, navigate back to the `Device Manager` and click the dropdown for other devices. The warning label should now be gone and Pytrack/Pysense should be installed.



Firmware Update Tools

We strongly recommend you to upgrade your firmware to the latest version as we are constantly making improvements and adding new features to the devices.

Here are the download links to the update tool. Please download the appropriate one for your OS and follow the instructions on the screen.

- Windows
- macOS (10.11 or Higher)
- Linux (requires `dialog` and `python-serial` package)

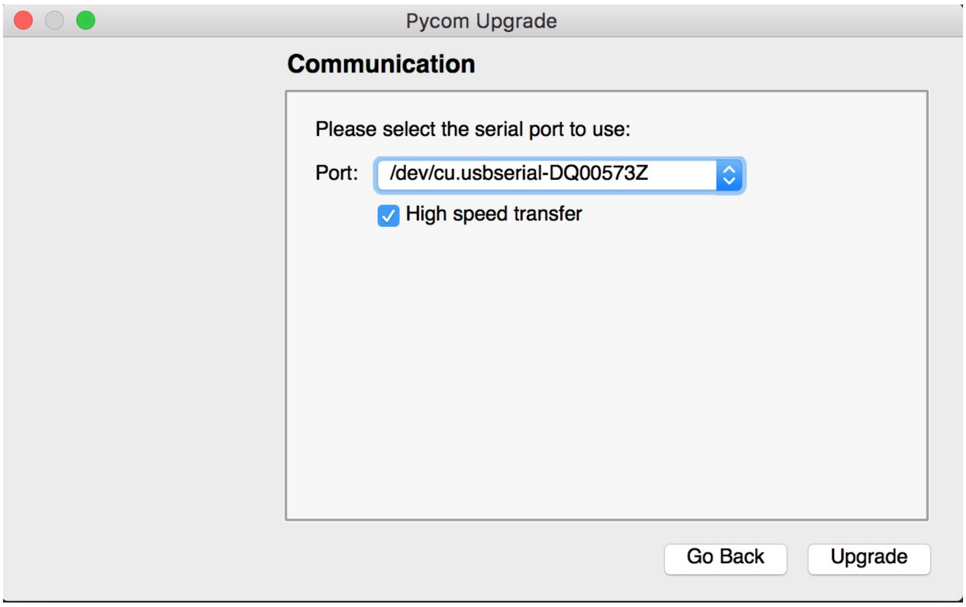
Previous versions of firmware are available for download **here**.

Updating Device Firmware

The basic firmware upgrade procedure can be found below, please follow these steps carefully:

- Expansion Board 2.0
- Pysense/Pytrack/Pyscan/Expansion Board 3.0

1. Disconnect your device from your computer
2. Insert module into the Expansion Board
3. Connect a jumper cable or wire between `G23` and `GND`
4. Reconnect the board via USB to your computer, this puts the device in ‘firmware update mode’.
5. Run the Firmware Upgrade tool

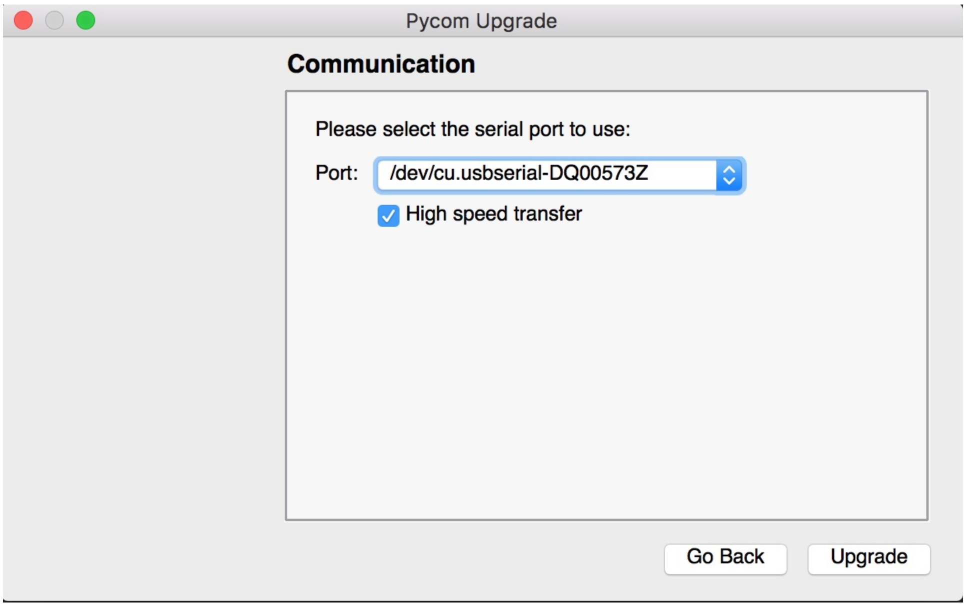


6. Remove the `G23` to `GND` jumper cable/wire
7. Reboot the device (button or power off then on), your device is now ready to use

If you are having any issues, make sure the **TX and RX jumpers** are present on your Expansion Board, as the jumpers sometimes come loose in the box during transport. Without these jumpers, the updater will fail.

When using a Pysense/Pytrack/Pyscan/Expansion Board 3.0 to update your module you are not required to make a connection between `G23` and `GND`, the Pysense/Pytrack/Pyscan/Expansion Board 3.0 will do this automatically.

1. Before connecting your module to a Pysense/Pytrack board, you should update the firmware on the Pysense/Pytrack. Instructions on how to do this can be found here.
2. Disconnect your device from your computer
3. Insert module into Expansion Board
4. Reconnect the board via USB to your computer
5. Run the Firmware Upgrade tool



6. Disconnect the USB cable from the board and reconnect it, your device is now ready to use

After you’re done with upgrading, you can use the Pymakr Plugins to upload and run programs in your device.

Pymakr Plugins

To make it as easy as possible Pycom has developed a plugin for two popular text editors, called Pymakr. These plugins have been built and are available for the following platforms:

ATOM

VS Code

Using your module

Now that you have connected and updated your pycom module and installed all the required software on your computer, we can begin programming your Pycom module.

If this is your first time using a Pycom module we highly recommend you read through the following pages:

- **Introduction to MicroPython:** This page will explain what Micropython is and its relation to Python.
- **MicroPython Examples:** We also recommend you browse these short MicroPython examples to familiarise yourself with its syntax. This is not meant as a comprehensive guide to MicroPython programming but rather a reference to those who already know programming. If you are new to python, or programming all together, we highly recommend searching the internet for Python tutorials. There are many very good tutorials available for free and the skills you learn will be easily transferable to our platform.
- **Your first Pymakr project:** Once you understand what MicroPython is, this guide will take you through setting up your first Pymakr project to blink the on-board RGB LED. This guide will explain the structure of a MicroPython project as well as how to upload it to your module.

Once you are familiar with MicroPython and Pymakr, the recommended way of uploading code to your module, you can explore the pages below. These will discuss in greater detail the various mechanisms for running code on your device as well as how to recover it if something goes wrong.

- **REPL:** The REPL (Read Evaluate Print Loop) is an interactive terminal that allows you to type in and test your code directly on the device, just like interactive python interpreter. It can be accessed via UART or Telnet. This is accessed easiest by using Pymakr but if you wish to use other tools, this page will explain how.
- **FTP:** All Pycom modules start up with a WiFi access point enabled, and a simple FTP server running on it. Once connected to the WiFi network, you can use FTP to transfer files over to your device wirelessly. This can be very useful if you do not have physical access to your device.
- **Safe Boot:** It is possible that some code you upload to your module will prevent you accessing the REPL or FTP server, preventing you from updating your scripts. This guide will detail how to safe boot your module and how to remove the offending scripts

from it.

Introduction to MicroPython

Our boards work with MicroPython; a Python 3.5 implementation that is optimised to run on micro controllers. This allows for much faster and more simple development process than using C.

Booting into MicroPython

When booting, two files are executed automatically: first **`boot.py`** and then **`main.py`**. These are placed in the `/flash` folder on the board. Any other files or libraries can be placed here as well, and can be included or used from `boot.py` or `main.py`.

The folder structure in `/flash` looks like the picture below. The files can be managed either using FTP or using the Pymakr Plugin.

cert		Directory
lib		Directory
sys		Directory
boot.py	1734	Python
main.py	14	Python

Tips & Tricks

Micropython shares majority of the same syntax as Python 3.5. The intention of this design is to provide compatibility upwards from Micropython to Python 3.5, meaning that code written for Micropython should work in a similar manner in Python 3.5. There are some minor variations and these should taken viewed as implementation differences.

Micropython also has a number of Micropython specific libraries for accessing hardware level features. Specifics relating to those libraries can be found in the Firmware API Reference section of this documentation.

> Micropython, unlike C/C++ or Arduino, **does not use braces {} to indicate blocks of code** specified for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is strictly enforced.
>
> The number of spaces in the indentation is variable but all statements within a block must be indented the same amount.

MicroPython Examples

To get you started with Python (MicroPython) syntax, we've provided you with a number of code examples.

Variable Assignment

As with Python 3.5, variables can be assigned to and referenced. Below is an example of setting a variable equal to a string and then printing it to the console.

```
variable = "Hello World"
print(variable)
```

Conditional Statements

Conditional statements allow control over which elements of code run depending on specific cases. The example below shows how a temperature sensor might be implemented in code.

```
temperature = 15
target = 10
if temperature > target:
    print("Too High!")
elif temperature < target:
    print("Too Low!")
else:
    print("Just right!")
```

Loops (For & While loop)

Loops are another important feature of any programming language. This allows you to cycle your code and repeat functions/assignments/etc.

`for` loops allow you to control how many times a block of code runs for within a range.

```
x = 0
for y in range(0, 9):
    x += 1
print(x)
```

`while` loops are similar to `for` loops, however they allow you to run a loop until a specific conditional is `true/false`. In this case, the loop checks if `x` is less than `9` each time the loop passes.

```
x = 0
while x < 9:
    x += 1
print(x)
```

Functions

Functions are blocks of code that are referred to by name. Data can be passed into it to be operated on (i.e. the parameters) and can optionally return data (the return value). All data that is passed to a function is explicitly passed.

The function below takes two numbers and adds them together, outputting the result.

```
def add(number1, number2):
    return number1 + number2

add(1, 2) # expect a result of 3
```

The next function takes an input name and returns a string containing a welcome phrase.

```
def welcome(name):
    welcome_phrase = "Hello, " + name + "!"
    print(welcome_phrase)

welcome("Alex") # expect "Hello, Alex!"
```

Data Structures

Python has a number of different data structures for storing and manipulating variables. The main difference (regarding data structures) between C and Python is that Python manages memory for you. This means there's no need to declare the sizes of lists, dictionaries, strings, etc.

Lists

A data structure that holds an ordered collection (sequence) of items.

```
networks = ['lora', 'sigfox', 'wifi', 'bluetooth', 'lte-m']
print(networks[2]) # expect 'wifi'
```

Dictionaries

A dictionary is like an address-book where you can find the address or contact details of a person by knowing only his/her name, i.e. keys (names) are associate with values (details).

```
address_book = {'Alex':'2604 Crosswind Drive','Joe':'1301 Hillview Drive','Chris':'323
6 Goldleaf Lane'}
print(address_book['Alex']) # expect '2604 Crosswind Drive'
```

Tuple

Similar to lists but are immutable, i.e. you cannot modify tuples after instantiation.

```
pycom_devices = ('wipy', 'lopy', 'sipy', 'gpy', 'fipy')
print(pycom_devices[0]) # expect 'wipy'
```

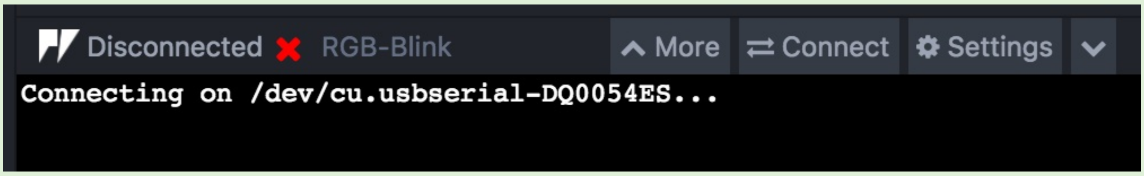
For more Python examples, check out these tutorials. Be aware of the implementation differences between MicroPython and Python 3.5.

Your First Pymakr Project

This guide will take you through how to setup your first project with Pymakr and make the on-board RGB LED flash various colours.

Creating a project in Pymakr

1. Firstly you will need to create a new, empty, directory on your computer. For this example we will create one called `RGB-Blink`.
2. Next you will need to open either Atom or Visual Studio Code depending on which you setup previously.
3. Once the text editor has loaded you will need to click `File` > `Open`, and open the directory you created in step 1

> If you are using Atom, it is important to check at this point that Atom has successfully identified the project. The name of the directory you created in step 1 (`RGB-Blink` in this case) should be shown in the Pymakr pane like so:
>
> 
>
>
> If this is not the case you can press `alt-ctrl-r` on Windows/Linux or `ctrl-alt-cmd-l` on macOS, in order to reload Atom and fix the issue.

4. Now that you have a project created, we need to add some files to it. A standard MicroPython project has the following structure:

```
RGB-Blink
|-lib
|  |- some_library.py
|-boot.py
|-main.py
```

- **`boot.py`** This is the first script that runs on your module when it turns on. This is often used to connect a module a a WiFi network so that Telnet and FTP can be used without connecting to the WiFi AP created by the module and not cluttering up the `main.py` file. As a beginner you do not need to use a `boot.py`.

  - **`main.py`** This script runs directly after `boot.py` and should contain the main code you wish to run on your device.
  - **`lib`** It is often a good idea to split out re-usable code into libraries. If you want to create or use libraries created by others, you will need to create a `lib` directory and put the library files in this. It is important that you put `.py` files directly into `lib` rather than creating a directory tree. By default MicroPython will not detect any libraries within sub-directories.

For this example, you will just need to create a `main.py` file.

Now that the project structure is setup, you may wish to configure project specific settings for Pymakr e.g. Which serial port to use. On Atom you need to click the `^` button on the Pymakr pane, then click `Project Settings`. On Visual Studio Code you need to click the `All commands` button on the bottom of the windows, then click `Pymakr > Project Settings`. This creates a file called `pymakr.conf` inside your project and populates it with default settings copied over from your global settings. A detailed explanation of these settings can be found here.

Controlling the on-board LED

Now that you have setup and configured your project, we can move on to programming your module. The first thing we will need to do is import some libraries in order to interact with the on-board LED. The Pycom firmware comes with a large amount of libraries for standard functionality built-in. You can find out more about these in the API documentation. For this example you will need to open the `main.py` file and add the following code:

```
import pycom
import time
```

This will import two libraries, `Pycom` which is responsible for Pycom specific features, such as the on-board LED and `time` which is a standard library used timing and delays.

You may have noticed that when you power up your Pycom module, the on-board LED blinks blue on a regular basis. This "heartbeat" is used as a way of know that your module has powered up and started correctly. Before we can change the colour of this LED we need to disable this heart beat. Below your imports you will need to add the following:

```
pycom.heartbeat(False)
```

Now it's time to test your code. On the Pymakr pane/bottom of the window you will see a `run` button. (If you haven't connected to your device yet, you will need to do that first). When you click the run button, the code in the currently open file will be executed on the device, but it won't copy it to the device. After running this code, you should see that that on-board LED stops blinking blue.

Now that we can confirm the device is connected and Pymakr is able to run code on it, we can complete our script to blink the LED like so:

```
import pycom
import time

pycom.heartbeat(False)

while True:
    pycom.rgbled(0xFF0000)  # Red
    time.sleep(1)
    pycom.rgbled(0x00FF00)  # Green
    time.sleep(1)
    pycom.rgbled(0x0000FF)  # Blue
    time.sleep(1)
```

Once you run the above script, it will run forever. You will notice this prevents you from accessing the interactive REPL on the device (You cannot see the `>>>` prompt). In order to stop the script, click onto the Pymakr terminal, and press `ctrl-c` on your keyboard. This should stop the script running and return you to the interactive REPL.

Uploading to your module

In the previous section we got code running on on your Pycom module using the `run` feature of Pymakr. This is useful for quick testing but has a couple of drawbacks. Firstly the code does not remain on the device permanently. If you reboot the device, it will no longer be running your code. Secondly, it will only work if you are using libraries built into the firmware. If you need any extra libraries, these need to be copied to the device first. This is where the `upload` feature comes in. If instead of `run` you click `upload`, Pymakr will upload all the files in the project (so long as their type is in the `sync_file_types` setting for your project). These then persist on your device even between reboots, and allows you to use libraries from the `lib` folder in your project.

If you need to remove files from your device you have two options, either connect via FTP and manage your files that way or format the device's internal flash like so:

```
import os
os.mkfs('/flash')
```

REPL (Read Evaluate Print Loop)

REPL stands for Read Evaluate Print Loop, and is the name given to the interactive MicroPython prompt that is accessible on the Pycom devices. Using the REPL is by far the easiest way to test out Python code and run commands. You can use the REPL in addition to writing scripts in `main.py` .

The following pages will explain how to use the REPL with both Serial USB and Telnet connections.

The REPL includes the following features:

- Input history: use arrow up and arrow down to scroll through the history
- Tab completion: press tab to auto-complete variables or module names
- Halt any executing code: with `Ctrl-C`
- Copy/paste code or output: `Ctrl-C` and `Ctrl-V`

There are a number of useful shortcuts for interacting with the MicroPython REPL. See below for the key combinations;

- `Ctrl-A` on a blank line will enter raw REPL mode. This is similar to permanent paste mode, except that characters are not echoed back.
- `Ctrl-B` on a blank like goes to normal REPL mode.
- `Ctrl-C` cancels any input, or interrupts the currently running code.
- `Ctrl-D` on a blank line will do a soft reset.
- `Ctrl-E` enters ‘paste mode’ that allows you to copy and paste chunks of text. Exit this mode using `Ctrl-D` .
- `Ctrl-F` performs a "safe-boot" of the device that prevents `boot.py` and `main.py` from executing

Serial USB REPL (UART)

To use the REPL, a Pycom device must be connected to the host computer with a USB connection either to an Expansion Board or to serial converter (a diagram of how to do this can be found the the getting started page for your module).

In order to connect to the REPL over USB serial, there are multiple methods. Detailed below are the explanations of how to do it in MacOS, Linux and Windows.

All platforms

By far the easiest way to access the USB UART REPL is via the our Pymakr plug-in for Atom and Visual Studio Code. This adds a pane to the bottom of the editors that allows you to directly access the REPL and any output from the device. Detailed instructions on how to setup Pymakr can be found here.

macOS and Linux

To open a serial USB connection from macOS, any serial tool may be used; in this example, the terminal tool `screen` will be used.

Open a terminal instance and run the following commands:

```
$ screen /dev/tty.usbmodem* 115200
```

Upon exiting `screen`, press `CTRL-A CTRL-\`. If the keyboard does not support the `\`-key (i.e. an obscure combination for `\` like `ALT-SHIFT-7` is required), the key combination can be remapped for the `quit` command:

- create `~/.screenrc`
- add bind `q` to the `exit` command

This will allow screen to exited by pressing `CTRL-A Q`.

> On Linux, `picocom` or `minicom` may be used instead of `screen`. The usb serial address might also be listed as `/dev/ttyUSB01` or a higher increment for `ttyUSB`. Additionally, the elevated permissions to access the device (e.g. group uucp/dialout or use `sudo`) may be required.

Windows

A terminal emulator is needed to open the connection from Windows; the easiest option is to download the free program, PuTTY.

COM Port

To use PuTTY the serial port (COM port) in which the Pycom device is connected, must be located. In Windows, this information can be found from the 'Device Manager' program.

1. Open the Windows start menu and search for 'Device Manager'
2. The COM port for the Pycom device will be listed as 'USB Serial Device' or a similar name
3. Copy/Write down the associated COM port (e.g. `COM4`)

Using Putty

1. With PuTTY open, click on `Session` in the left-hand panel
2. Next click the `Serial` radio button on the right and enter the associated COM port (e.g. `COM4`) in the `Serial Line` box
3. Finally, click the `Open` button

Telnet REPL

Pycom devices also support a connection via `telnet`, using the device's on board WiFi/WLAN. Connect to the device's WiFi Access Point (AP) and using the following credentials to connect to the AP. The WiFi `SSID` will appear upon powering on a Pycom Device for the first time (e.g. `lopy-`). To re-enable this feature at a later date, please see network.WLAN.

- password: `www.pycom.io`

Telnet Server

Additionally, to use the MircoPython REPL over telnet, further authentication is required. The default credentials for the telnet server are:

- username: `micro`
- password: `python`

See network.server for info on how to change the default authentication.

All platforms

By far the easiest way to access the Telnet REPL is via the our Pymakr plug-in for Atom and Visual Studio Code. This adds a pane to the bottom of the editors that allows you to directly access the REPL and any output from the device. Detailed instructions on how to setup Pymakr can be found here.

macOS and Linux

Once the host machine is connected to the Pycom device's Access Point, a telnet connection may be opened from a terminal instance.

```
$ telnet 192.168.4.1
```

Upon connection, the telnet program will prompt for the **`username`** and **`password`** in the section above.

Windows

A terminal emulator is needed to open a telnet connection from Windows; the easiest option is to download the free program, PuTTY.

1. With PuTTY open, select telnet as connection type and leave the default port (`23`)
2. Next enter the IP address of the Pycom device (e.g. `192.168.4.1`)
3. Finally click `Open`

> When using a Pycom device with a personal, home or office WiFi access point, the telnet connection may still be used. In this instance, the user will need to determine the Pycom device's local IP address and substitute this for `192.168.4.1` , referred to in the earlier sections.

FTP (Local File System)

There is a small internal file system accessible with each Pycom device, called `/flash`. This is stored within the external serial flash memory. If a microSD card is also connected and mounted, it will be available as well. When the device starts up, it will always boot from the `boot.py` located in the `/flash` file system.

The file system is accessible via the native FTP server running on each Pycom device. Open an FTP client and connect to:

- url: `ftp://192.168.4.1`
- username: `micro`
- password: `python`

See network.server for information on how to change the defaults. The recommended clients are:

- macOS/Linux: default FTP client
- Windows: Filezilla and FireFTP

For example, from a macOS/Linux terminal:

```
$ ftp 192.168.4.1
```

The FTP server doesn't support active mode, only passive mode. Therefore, if using the native unix FTP client, immediately after logging in, run the following command:

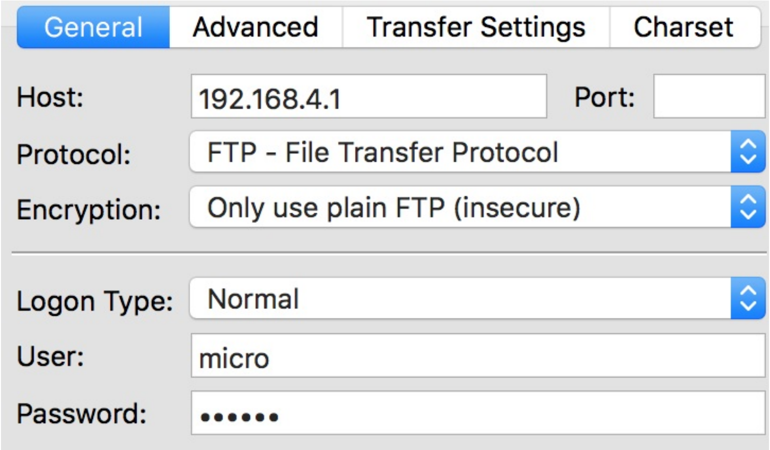
```
ftp> passive
```

The FTP server only supports one connection at a time. If using other FTP clients, please check their documentation for how to limit the maximum allowed connections to one at a time.

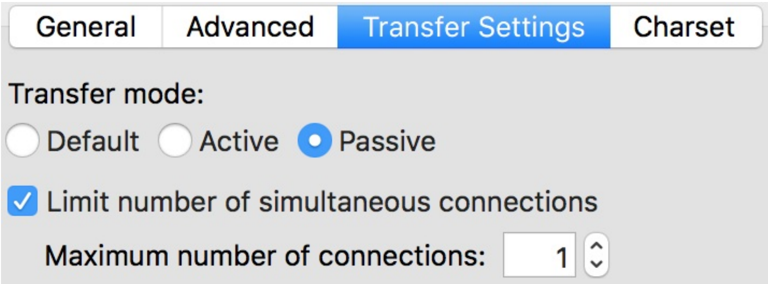
FileZilla

If using FileZilla, it's important to configure the settings correctly.

Do not use the quick connect button. Instead, open the site manager and create a new configuration. Within the `General` tab, ensure that encryption is set to: `Only use plain FTP (insecure)`.



In the `Transfer Settings` tab, limit the max number of connections to one. Other FTP clients may behave in a similar ways; visit their documentation for more specific information.



Boot Modes

If powering up normally or upon pressing the reset button, a Pycom module will boot into standard mode; the `boot.py` file will be executed first, followed by `main.py`. It is possible to alter the boot procedure of the module by tying certain pins `high` or `low` when the module boots.

Bootloader

If you updated your device before using it, you have already put the device into bootloader mode. This is achieved by connecting `G23` to `GND` while the device boots. If you used a Pysense/Pytrack to update, it did this automatically for you. You only need to put your Pycom module into bootloader mode if you are updating its firmware, or are programming your own low level code. This is not required if you are updating your MicroPython code.

Safe Boot

Some times the code you have written will prevent you gaining access to the REPL or prevent you updating your code. Some example may be:

- You disabled the WiFi/UART
- Your code gets stuck before reaching the REPL
- You set a socket as blocking but never receive any data

In order to fix this you can safe boot your module. This will prevent `boot.py` and `main.py` from being executed and will drop you straight into the interactive REPL. After reset, if `P12` pin is held `high` (i.e. connect it to the `3V3` output pin), the heartbeat LED will begin flashing orange slowly. If after 3 seconds the pin is still held high, the LED will start blinking faster. In this mode the module will do the same as previously explained but it will also select the previous OTA image to boot if you have updated the module via the OTA update procedure (updates performed via the firmware update tool do not count). This is useful if you flashed a OTA update that breaks the device.

Pin `P12` released during:

1st 3 secs window	2nd 3 secs window
Disable `boot.py` and `main.py`	same as previous but using previous OTA firmware

The selection made during safe boot is not persistent, therefore after the next normal reset, the latest firmware will proceed to run again.

If problems occur within the filesystem or you wish to factory reset your module to remove your code, run following code in the REPL:

```
>>> import os
>>> os.mkfs('/flash')
```

Be aware, resetting the flash filesystem will delete all files inside the internal device storage (not the SD card) and they cannot be recovered.

Reset

Pycom devices support both soft and hard resets. A soft reset clears the state of the MicroPython virtual machine but leaves hardware peripherals unaffected. To do a soft reset, press `Ctrl+D` on the REPL or from within a script, run:

```
>>> import sys
>>> sys.exit()
```

A hard reset is the same as performing a power cycle to the device. In order to hard reset the device, press the `reset` switch or run:

```
>>> import machine
>>> machine.reset()
```

Registering a Pycom Device

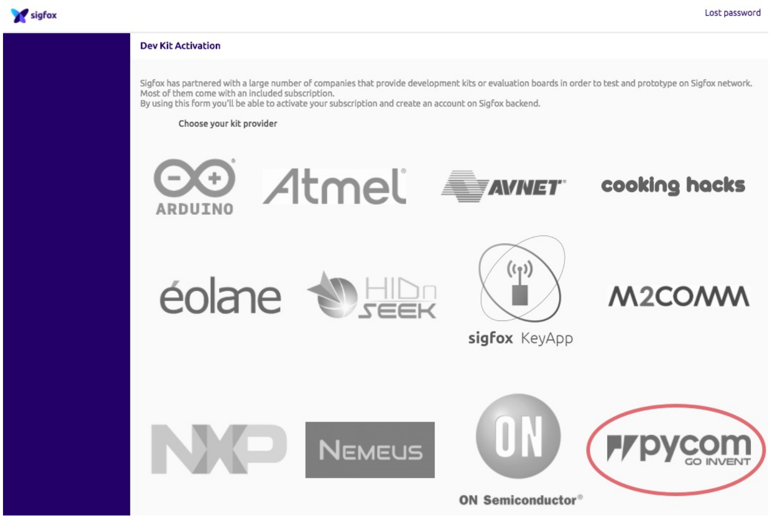
Some of our devices require registration before you can utilise specific features such as certain types of networking. Please see the list below for setup guides to ensure that your device is registered and activated on the various platforms required to access all of the available features.

Not all Pycom devices require activation; most features work immediately out of the box!

Registering with Sigfox

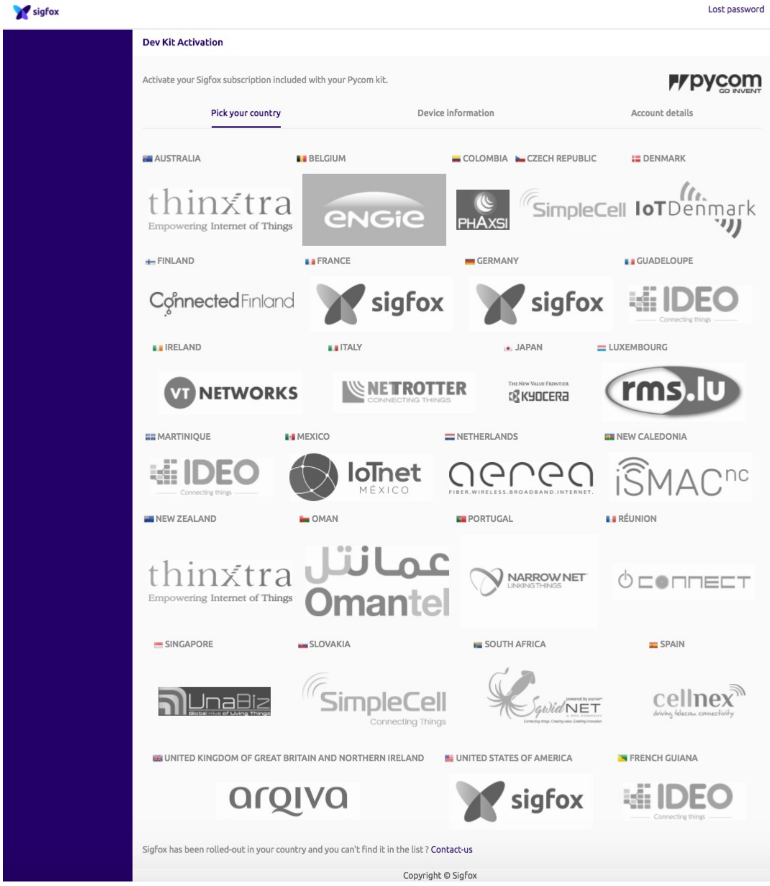
To ensure the device has been provisioned with **Device ID** and **PAC number**, please update to the latest firmware.

In order to send a Sigfox message, the device need to register with the Sigfox Backend. Navigate to https://backend.sigfox.com/activate to find the list of Sigfox enabled development kits.

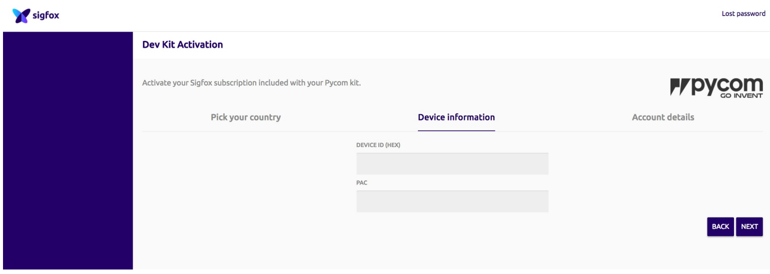


Select `Pycom` to proceed.

Next choose a Sigfox Operator for the country where the device will be activated. Find the specific country and select the operator to continue.



Now need to enter the device's **Device ID** and **PAC number**.

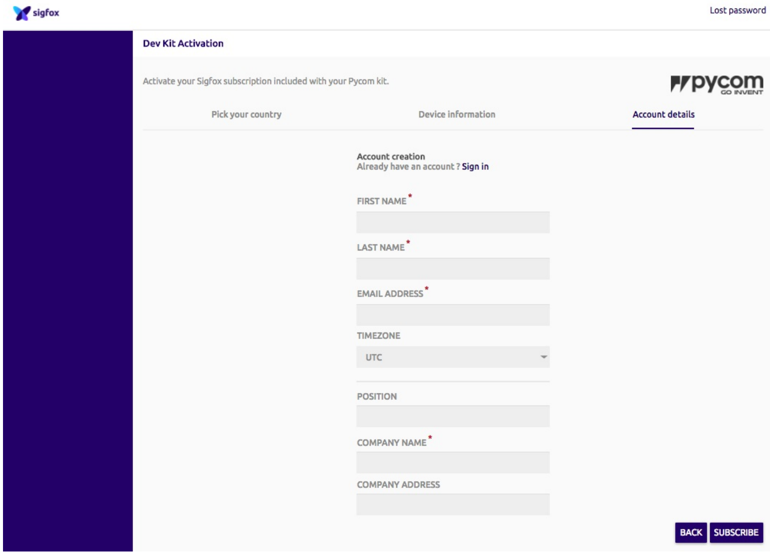


The **Device ID** and **PAC number** are retrievable through a couple of commands via the REPL.

```
from network import Sigfox
import binascii

# initalise Sigfox for RCZ1 (You may need a different RCZ Region)
sigfox = Sigfox(mode=Sigfox.SIGFOX, rcz=Sigfox.RCZ1)

# print Sigfox Device ID
print(binascii.hexlify(sigfox.id()))

# print Sigfox PAC number
print(binascii.hexlify(sigfox.pac()))
```

See `Sigfox` for more info about the Sigfox Class and which RCZ region to use.

Once the device's Device ID and PAC number have been entered, create an account. Provide the required information including email address and click to continue.



An email confirming the creation of a Sigfox Backend account and the successful registration of the device should arrive at the users inbox.

Cellular registration

In order to use your GPy/FiPy on a cellular network you are required to get a SIM card from a local provider. *Note:* This might differ from a standard SIM you can buy in a store, our devices do not support standard LTE.

Currently we are not able to provide any specific details about how to get such a SIM card and how to register it as most deployments are closed trials, each carrier has it's own rules (for example, whether they require special SIMs or not).

We recommend contacting your local cellular providers to check their plans surrounding LTE CAT-M1 and NB-IoT. By contacting them, you will show the carriers that there is local interest in deploying such networks.

You can find a map of deployed networks and open labs here.

LoRaWAN Registration

Raw LoRa

When using raw LoRa, you do not have to register your module in any way. The modules can talk to each other directly.

LoRaWAN

In order to connect your LoRa capable Pycom module to a LoRaWAN network you will have to register your device with the desired network. We are unable to provide instructions for all LoRaWAN networks but below you will find some generic instructions, along with links to any specific guides we are aware of.

Generic instructions

Firstly you will need to get your modules `Device EUI`, this can be achieved using the following code:

```
from network import LoRa
import ubinascii

lora = LoRa(mode=LoRa.LORAWAN)
print(ubinascii.hexlify(lora.mac()).upper().decode('utf-8'))
```

The output will be a hex string like: `70B3D5499585FCA1`. Once you have this you will need to provide it to your LoRaWAN network which will then provide you with the details need to connect via Over-the-Air Activation (OTAA) or Activation by Personalisation (ABP)

OTAA

If you wish to connect via OTAA (which is the recommended method) the network will provide you with an `Application EUI` and `Application Key`. The former identifies what application your device is connecting to, the latter is a shared secret key unique to your device to generate the session keys that prove its identity to the network. Once you have these you can use the LoRaWAN OTAA example code to connect to the network.

ABP

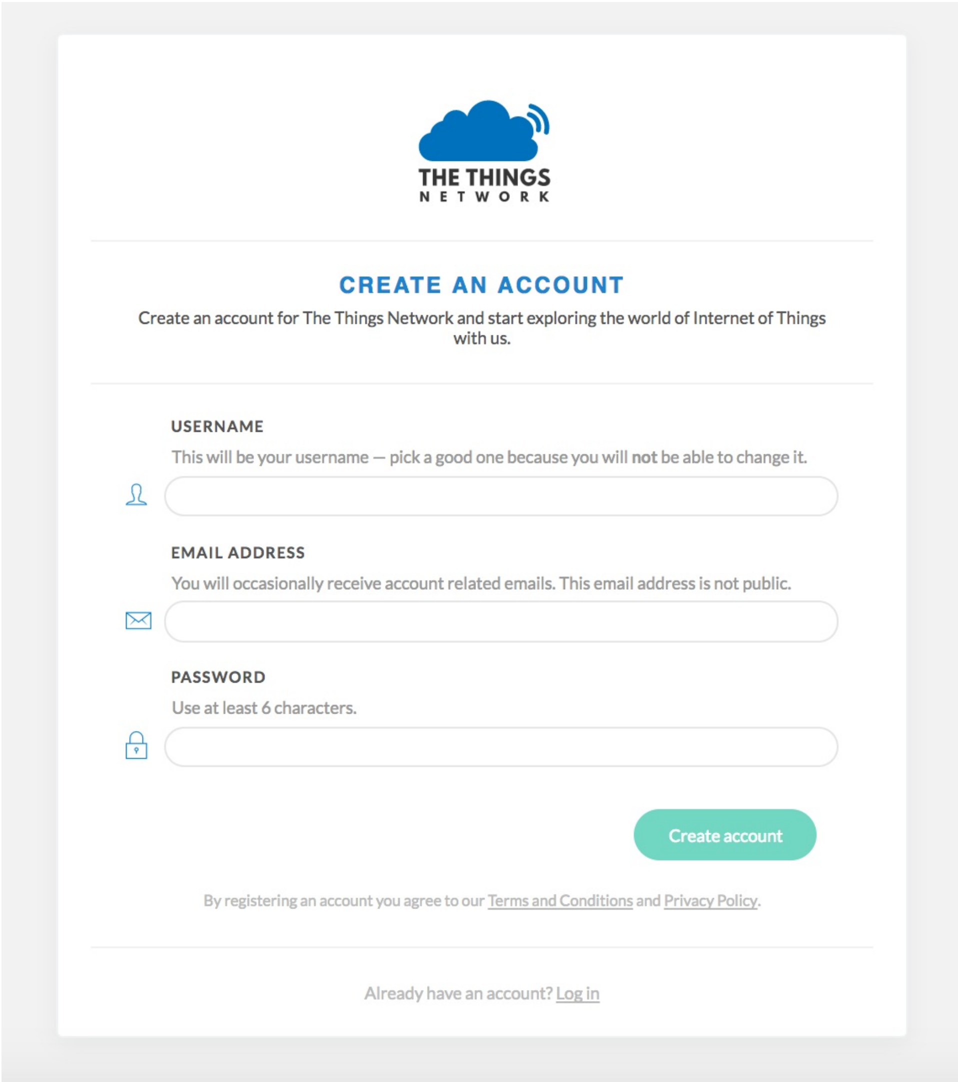
With ABP the encryption keys enabling communication with the network are preconfigured in the device. The network will need to provide you with a `Device Address`, `Network Session Key` and `Application Session Key`. Once you have these you can use the LoRaWAN ABP example code to connect to the network.

Networks

If you cannot find your favourite LoRaWAN network in the list above, please consider writing a tutorial for how to connect a Pycom module with it and contribute it to this documentation via a GitHub pull request.

The Things Network

In order to use The Things Network (TTN) you should navigate to their website and create/register an account. Enter a username and an email address to verify with their platform.

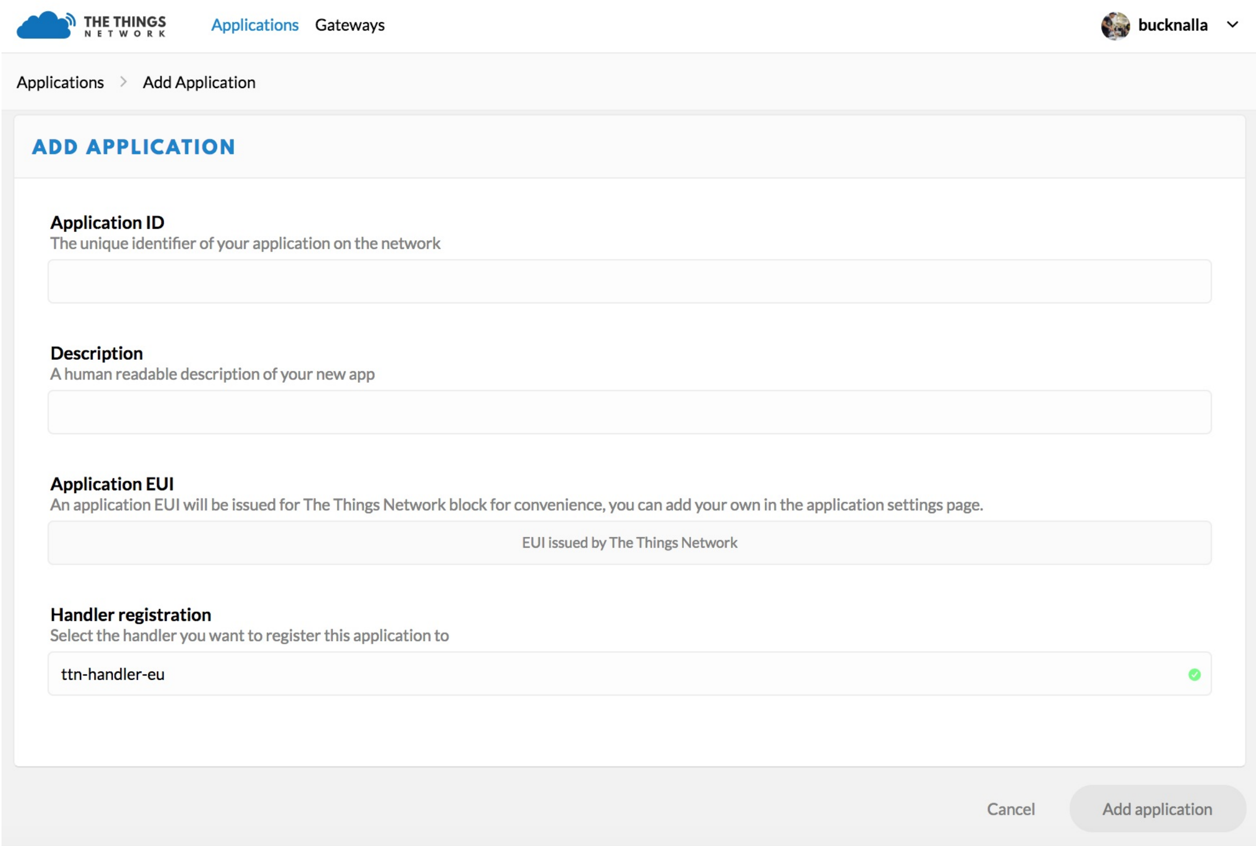


Once an account has been registered, you can register your Pycom module as either a node or a nano-gateway. The steps below will detail how to do this.

Create an application

In order to register your device to connect to the things network, you must first create an application for these devices to belong to. This way the Network will know where to send the devices data to.

Selecting the `Applications` tab at the top of the TTN console, will bring up a screen for registering applications. Click register and a new page, similar to the one below, will open.



Enter a unique `Application ID` as well as a Description & Handler Registration.

Now the Pycom module nodes can be registered to send data up to the new Application.

Register a Device

To connect nodes to a things network gateway, devices need to be added to the application. To do this, navigate to the `Devices` tab on the `Application` home page and click the `Register Device` button.

In the `Register Device` panel, complete the forms for the `Device ID` and the `Device EUI`. The `Device ID` is user specified and is unique to the device in this application. The `Device EUI` should be a globally unique identifier for the device. You can run the following on you Pycom module to retrieve its EUI.

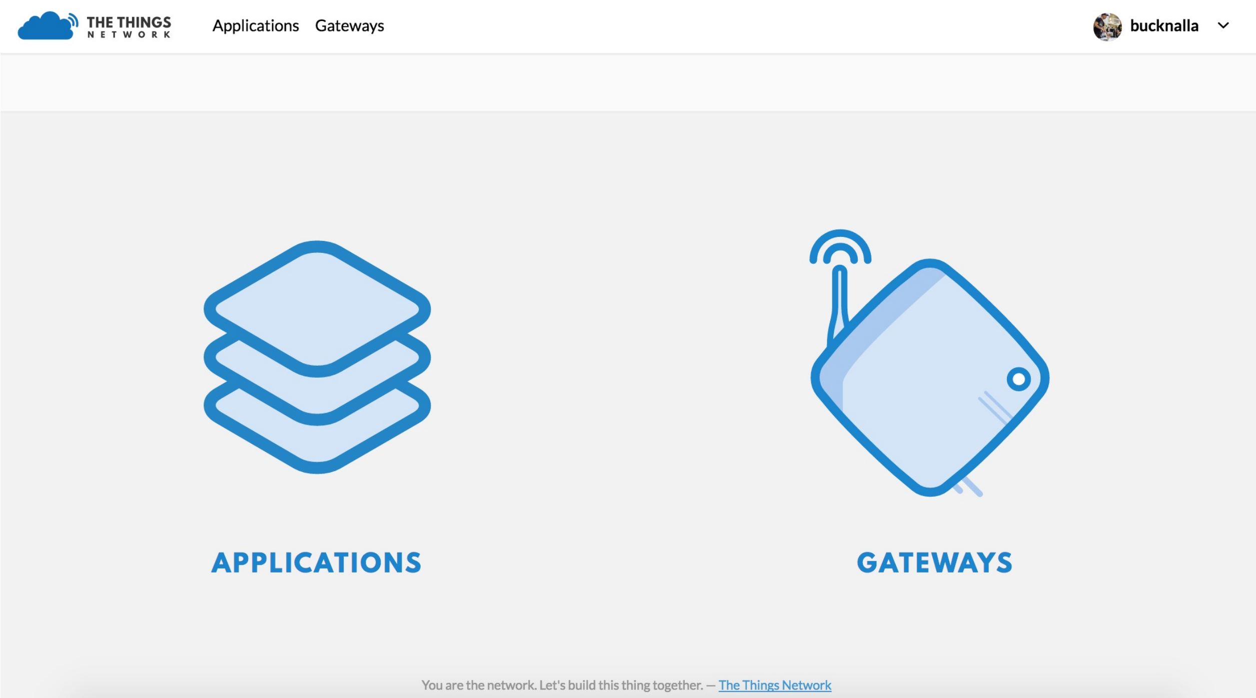
```
from network import LoRa
import ubinascii

lora = LoRa()
print("DevEUI: %s" % (ubinascii.hexlify(lora.mac()).decode('ascii')))
```

Once the device has been added, change the `Activation Method` between `OTAA` and `ABP` depending on user preference. This option can be found under the `Settings` tab.

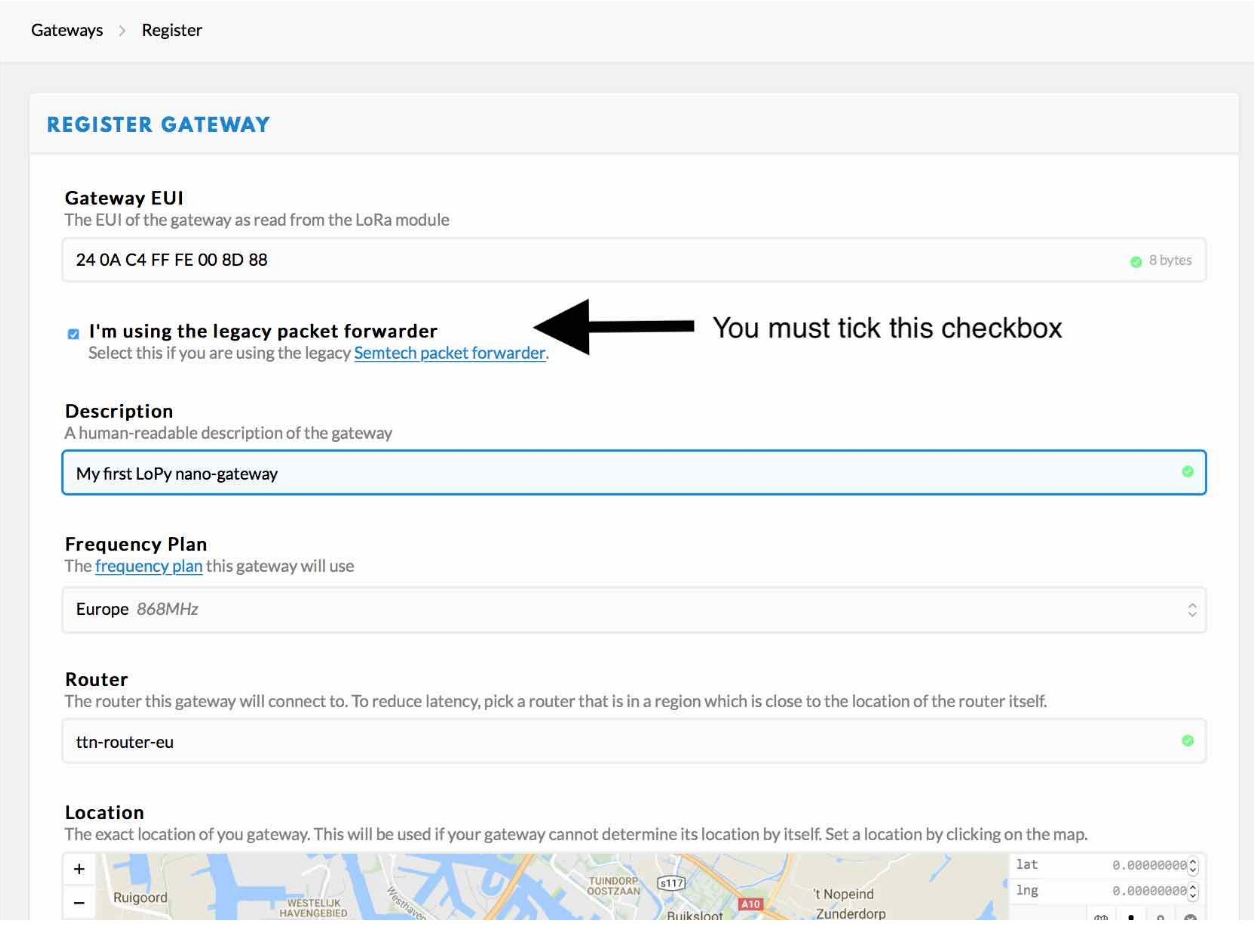
Register a Nano-Gateway

You can also setup your Pycom module to act as a gateway with The Things Network. The code required to do this can be found here.

Inside the TTN Console, there are two options, `Applications` and `Gateways`. Select `Gateways` and then click on `register Gateway`. This will allow for the set up and registration of a new nano-gateway.



On the Register Gateway page, you will need to set the following settings:



These are unique to each gateway, location and country specific frequency. Please verify that correct settings are selected otherwise the gateway will not connect to TTN.

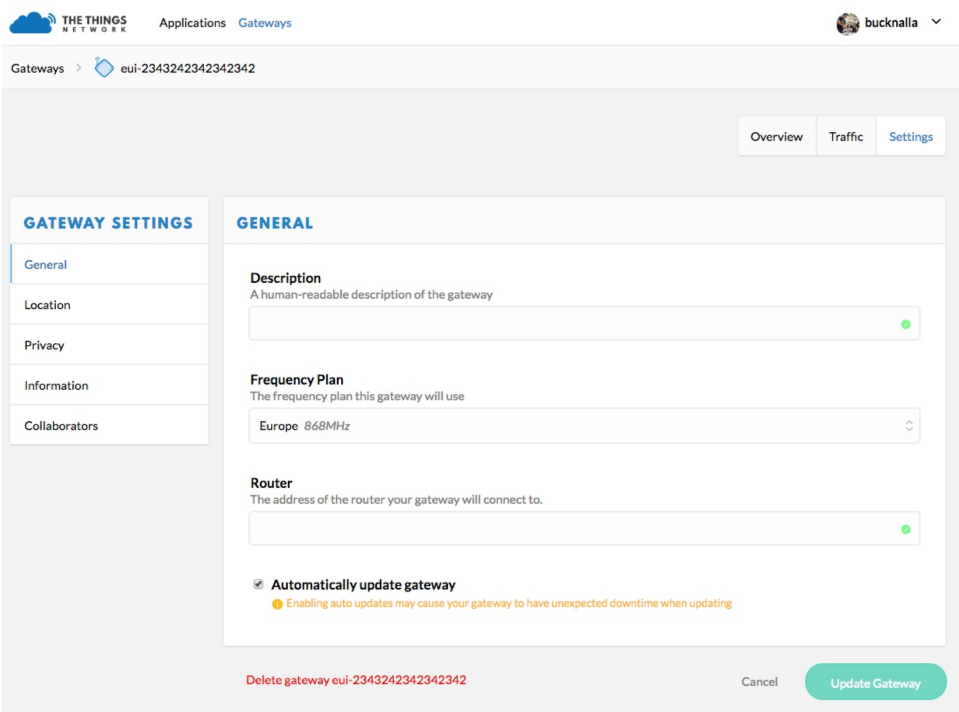
You need to tick the "I'm using the legacy packet forwarder" to enable the right settings. This is because the Nano-Gateway uses the 'de facto' standard Semtech UDP protocol.

Option	Value
Protocol	Packet Forwarder
Gateway EUI	User Defined (must match `config.py`)
Description	User Defined
Frequency Plan	Select Country (e.g. EU - 868 MHz)
Location	User Defined
Antenna Placement	Indoor or Outdoor

Most LoRaWAN network servers expect a Gateway ID in the form of a unique 64-bit hexadecimal number (called a EUI-64). The recommended practice is to produce this ID from your board by expanding the WiFi MAC address (a 48-bit number, called MAC-48). You can obtain that by running this code prior to configuration:

```
from network import WLAN
import binascii
wl = WLAN()
binascii.hexlify(wl.mac())[:6] + 'FFFE' + binascii.hexlify(wl.mac())[6:]
```

Once these settings have been applied, click `Register Gateway`. A Gateway Overview page will appear, with the configuration settings showing. Next click on the `Gateway Settings` and configure the Router address to match that of the gateway (default: `router.eu.thethings.network`).



The `Gateway` should now be configured.

Connecting to Objenious LoRaWAN 'Spot' network

Identifiers

To connect a Pycom LoRa device (LoPy, LoPy4, FiPy) to Objenious you'll need to provision it. This requires three pieces of information

- Device EUI (DevEUI)
- Application EUI (AppEUI)
- Application Key (AppKey)

Device EUI

This comes from the device itself and can be obtained from `lora.mac()` .
To obtain the required hexadecimal representation you can run the following code on your LoPy:

```
from network import LoRa
import ubinascii

lora = LoRa()
print("DevEUI: %s" % (ubinascii.hexlify(lora.mac()).decode('ascii')))
```

Application EUI and Application Key

Application EUI and Key are two LoRaWAN parameters that should ideally by generated by you, if supplying devices to end customers.
The Application EUI is a EUI-64 (8 bytes) identifier which should be universally unique - it's usually allocated from a MA-S block purchased from the IEEE Registration Authority.
The Application Key should be a randomly generated, secure, 128 bit (16 byte) token.

For testing purposes we provide a script which generates a random Application EUI from our assignment and a series of Application Keys:

EUI/Key generator for testing

(note: the Application EUI produced by this script is not guaranteed to be unique)

To use the script make sure you are using Python 3.6 on your computer and run it (on your computer, *not* on the Pycom board) as:

`python generate_keys.py 1`

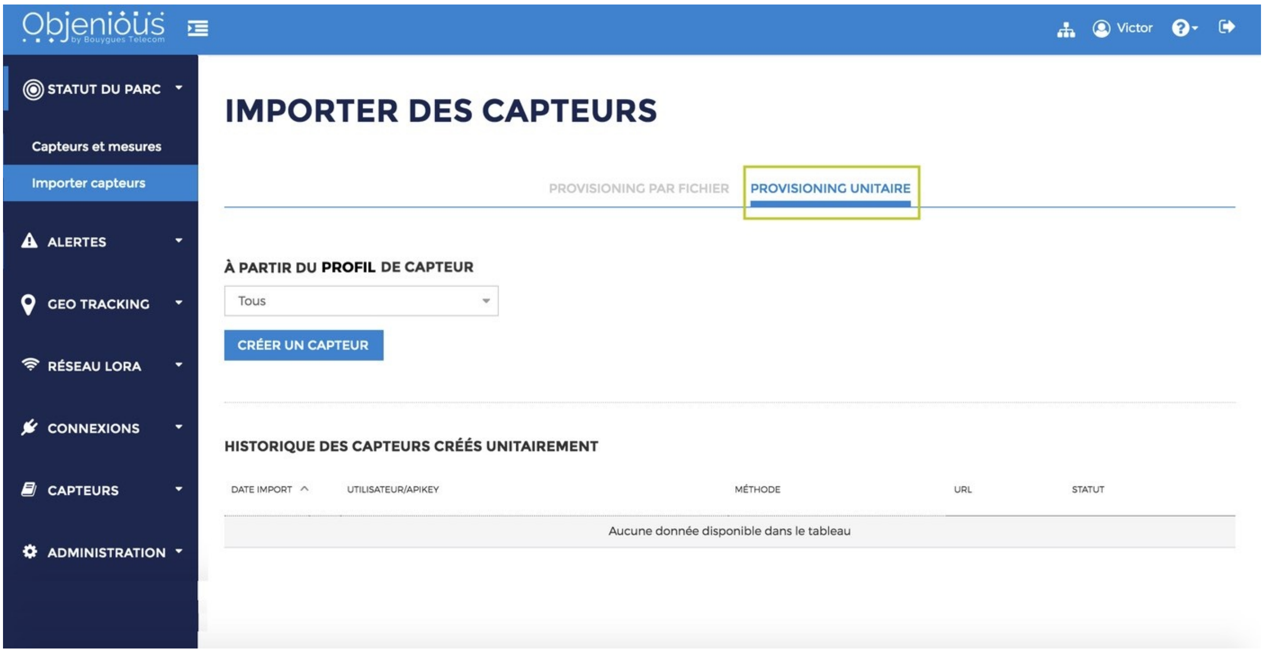
The output will be similar to:

```
AppEUI: 70b3d54923e36a89
AppKeys:
78fe712d96f46784a98b574a8cd616fe
```

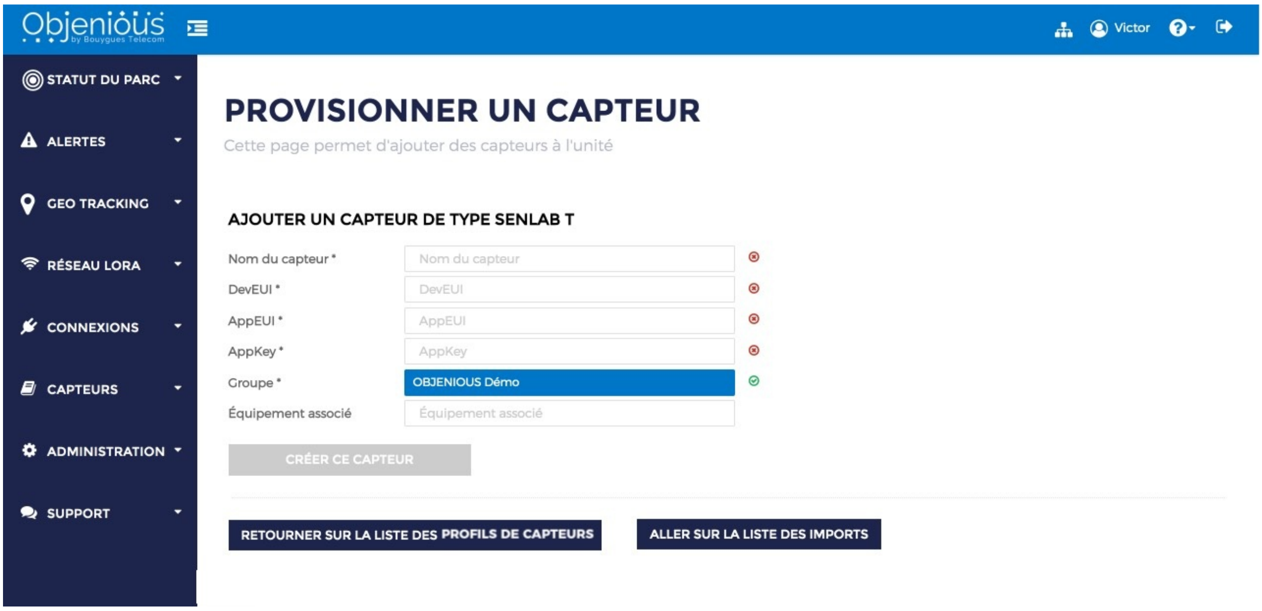
If you are registering multiple devices you can generate more Applications Keys by changing `1` to your desired number of devices.

Provisioning

Once you have the three identifiers for your device you need to register them on the Objenius portal.
Follow "Importer des capteurs" under "Statuc do Parc" and select "Provisioning Unitaire":



Once there give your device a name and enter the DevEUI, AppEUI and AppKey obtained from the steps above:


Objenious by Bouygues Telecom
Victor
STATUT DU PARC
ALERTES
GEO TRACKING
RÉSEAU LORA
CONNEXIONS
CAPTEURS
ADMINISTRATION
SUPPORT
PROVISIONNER UN CAPTEUR
Cette page permet d'ajouter des capteurs à l'unité
AJOUTER UN CAPTEUR DE TYPE SENLAB T
Nom du capteur *
Nom du capteur
DevEUI *
DevEUI
AppEUI *
AppEUI
AppKey *
AppKey
Groupe *
OBJENIOUS Démo
Équipement associé
Équipement associé
CRÉER CE CAPTEUR
RETOURNER SUR LA LISTE DES PROFILS DE CAPTEURS
ALLER SUR LA LISTE DES IMPORTS

Pymakr Plugins

To make it as easy as possible Pycom has developed a plugin for two popular text editors, called Pymakr. These plugins have been built and are available for the following platforms:

ATOM

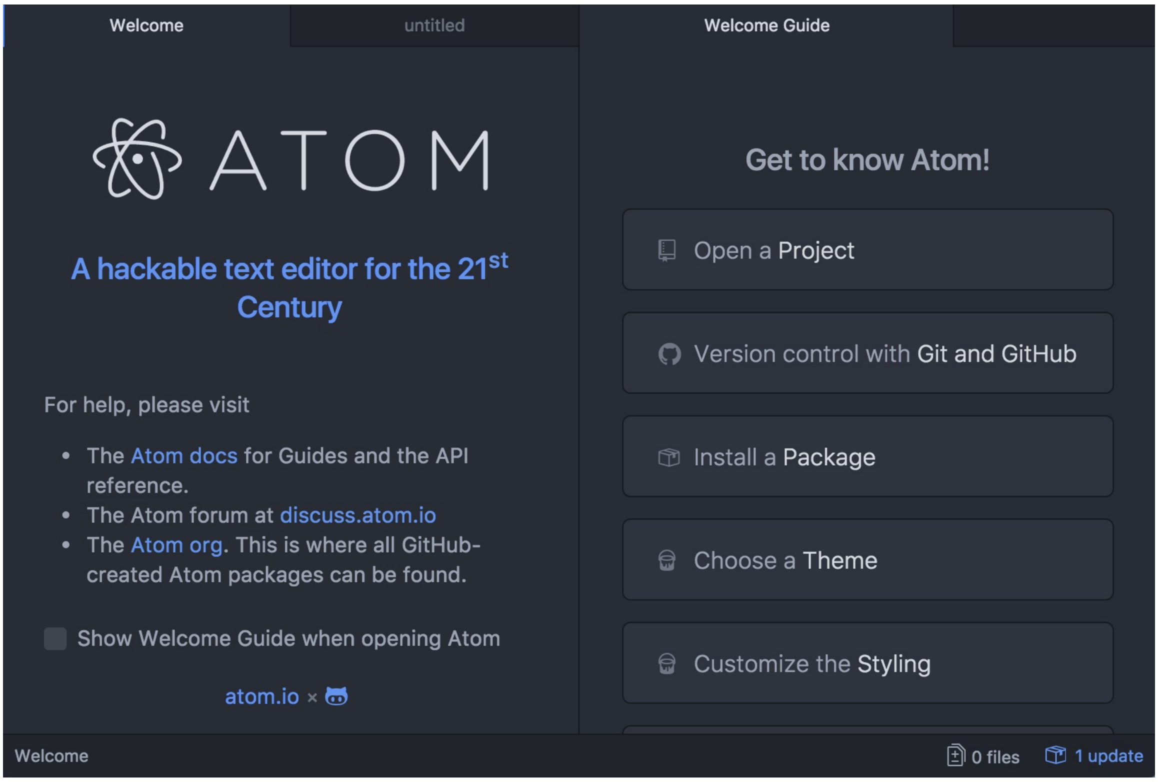
VS Code

Pymakr Plugin Installation for Atom

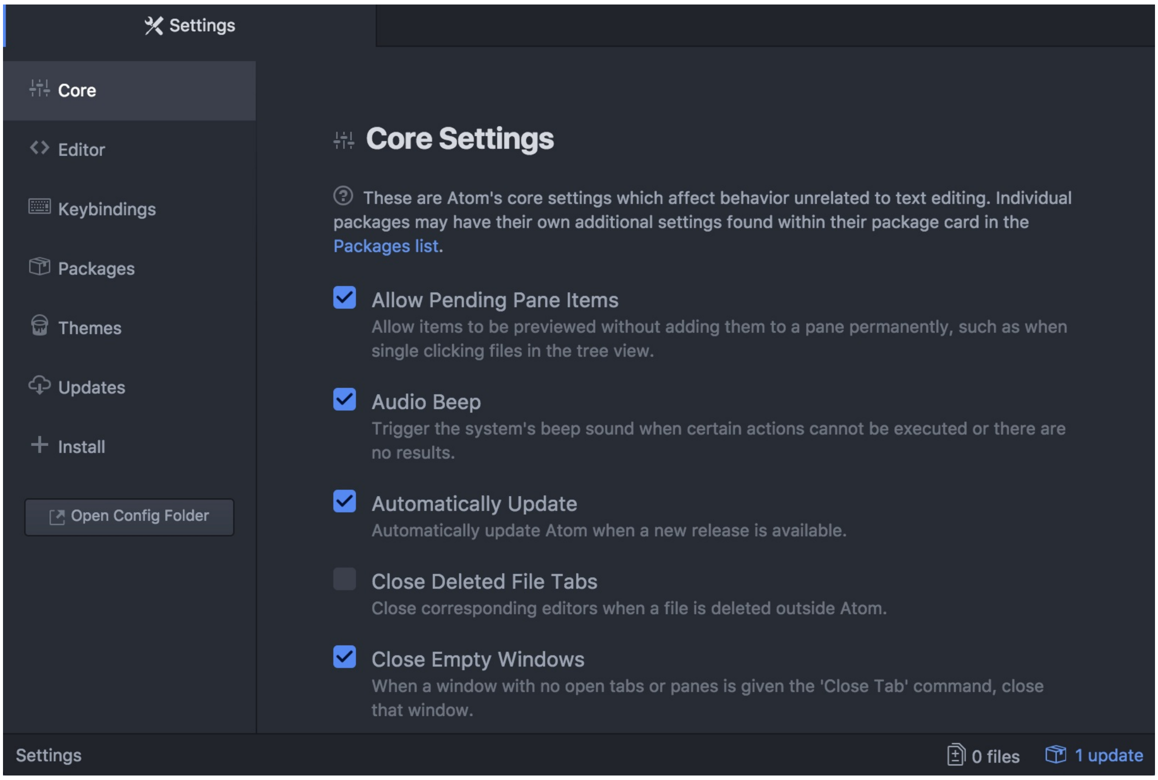
For beginners, users getting started with MicroPython & Pycom as well as Atom text editor users, we recommend the **Pymakr Plugin for Atom**. This section will help you get started using the Atom Text Editor & Pymakr Plugin.

Please follow these steps to install the Pymakr Plugin:

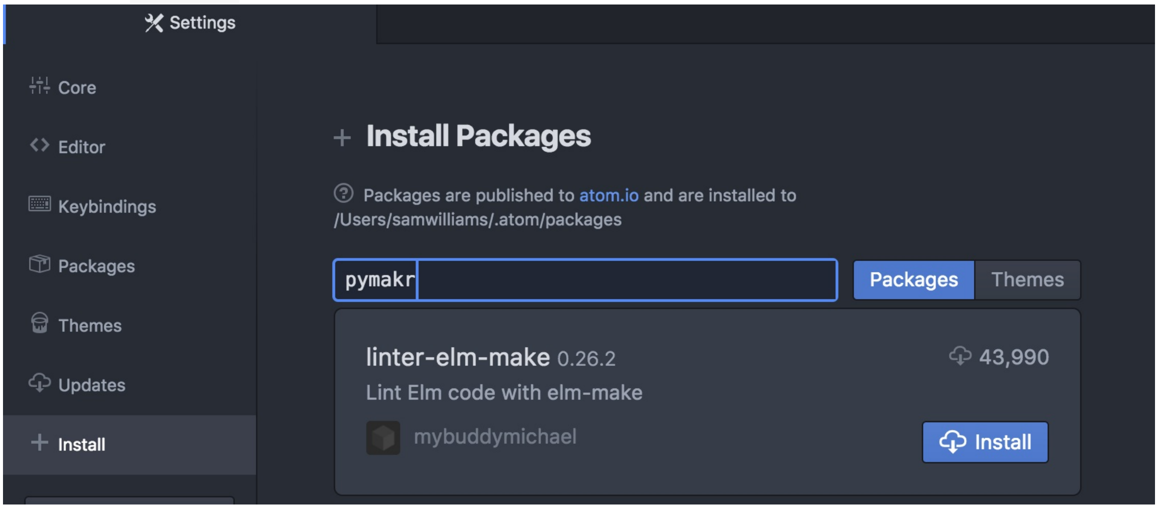
1. Ensure that you have Atom installed and open.



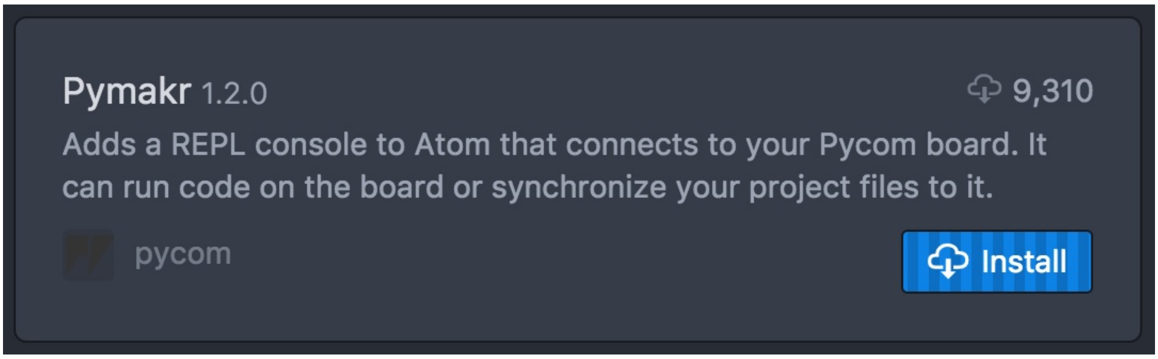
2. Navigate to the Install page, via `Atom > Preferences > Install`



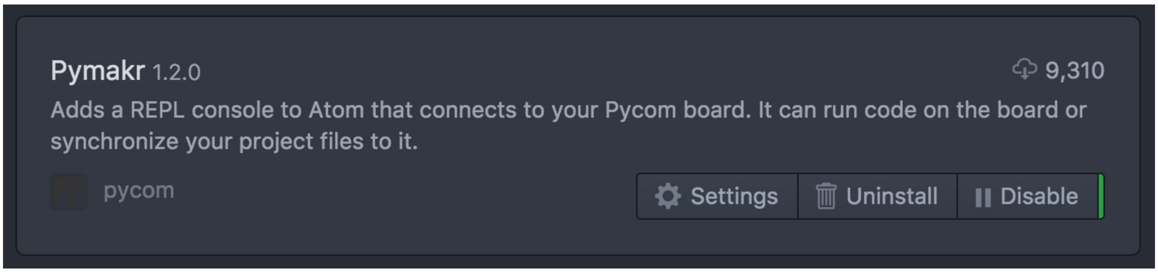
3. Search for `Pymakr` and select the official Pycom Pymakr Plugin.



4. You should now see and click the Install button. This will download and install the Pymakr Plugin.



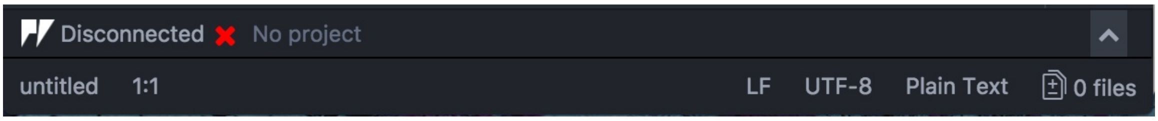
5. That's it! You've installed the Pymakr Plugin for Atom.



Connecting via Serial USB

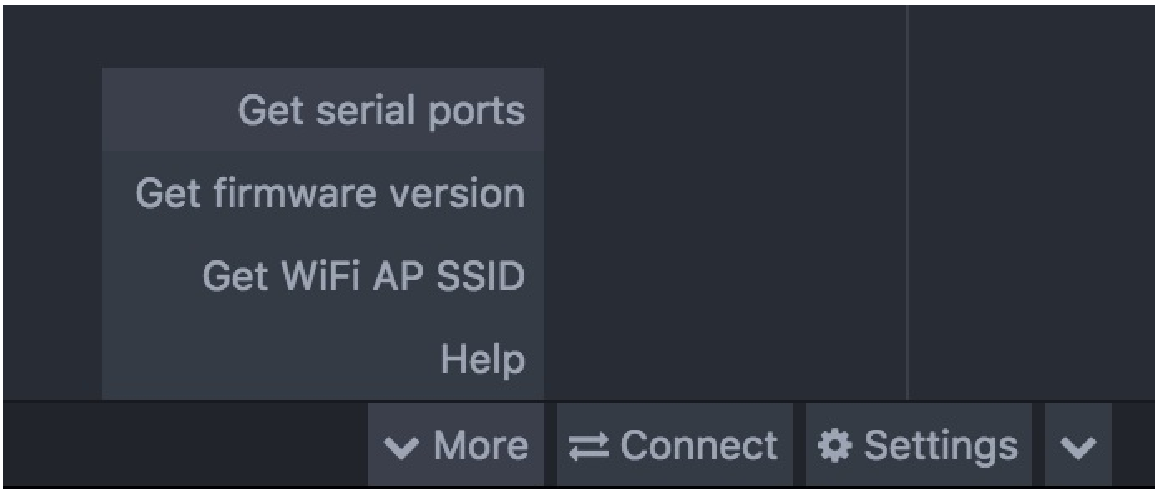
After installing the Pymakr Plugin, you need to take a few seconds to configure it for first time use. Please follow these steps:

1. Connect your Pycom device to your computer via USB. If you are using an Expansion Board 2.0, and have just finished a firmware upgrade, be sure to **remove the wire between GND and G23** and reset your device by pressing the button. Note: you don't need the wire for Expansion Board 3.0
2. Open Atom and ensure that the Pymakr Plugin has correctly installed.

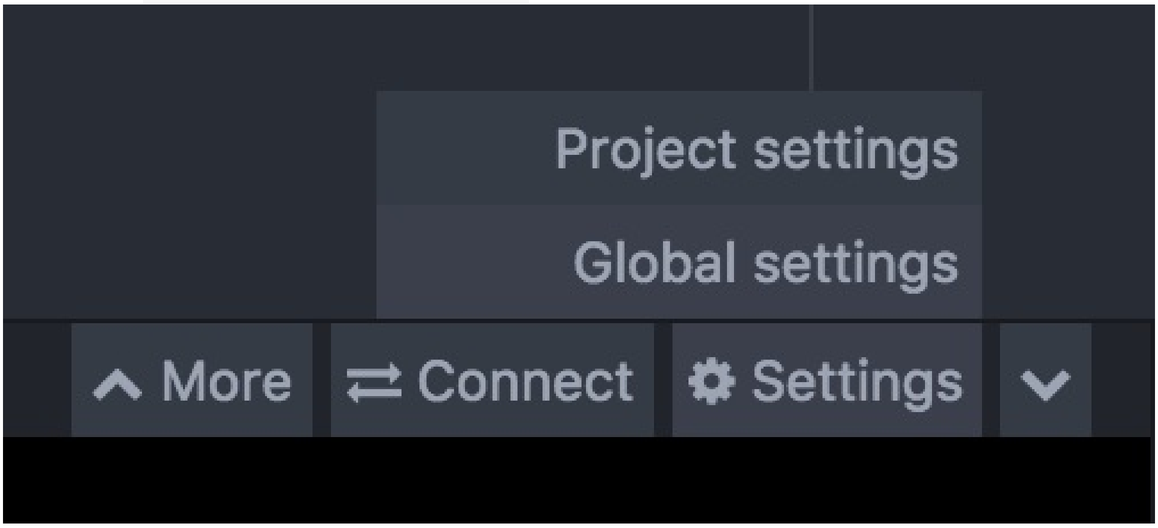


3. Open the Pymakr console by clicking the `^` button, located in the lower right side of the Atom window.

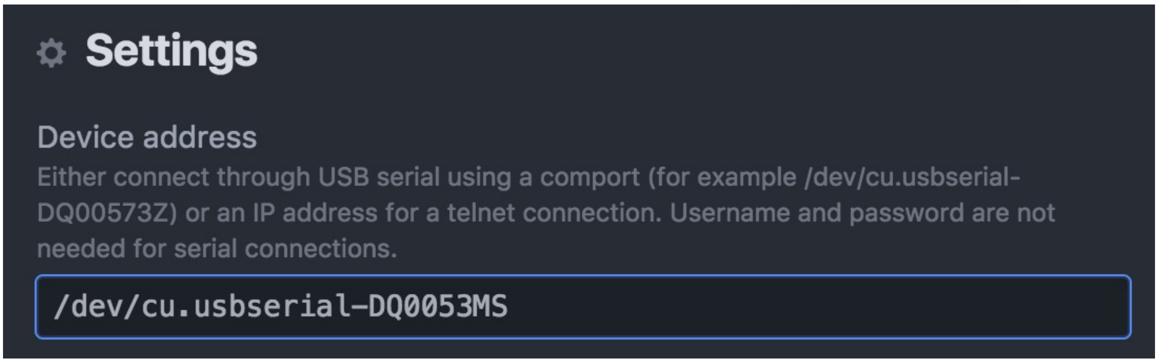
4. Click, `More` followed by `Get Serial Ports` . This will copy the serial address of your expansion board to your clipboard.



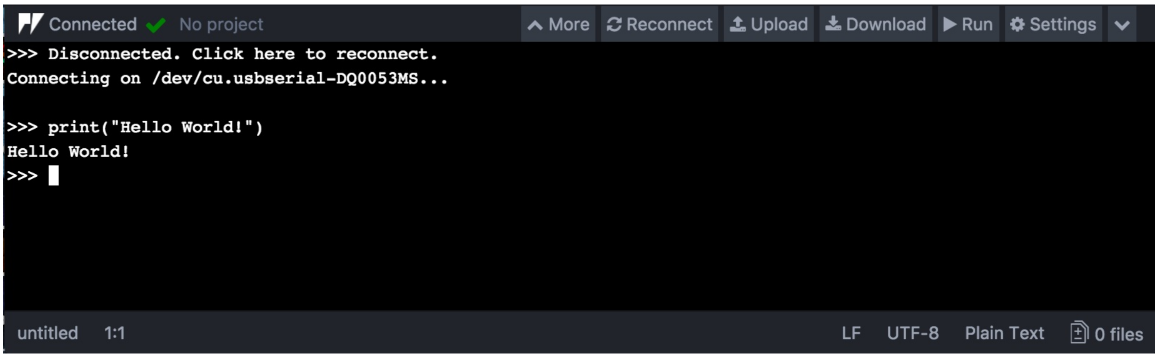
5. Navigate to `Settings > Global Settings`



6. Paste the serial address you copied earlier into the text field `Device Address`



7. Press connect and the Pymakr console should show three arrows `>>>`, indicating that you are connected



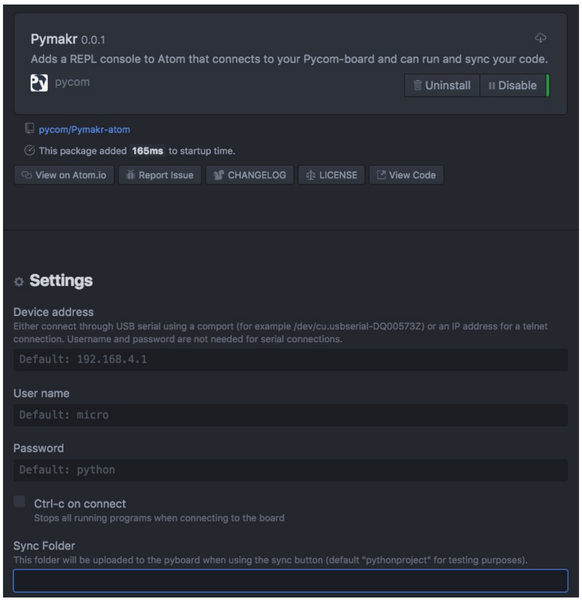
These settings can also be applied on a per project basis by clicking `Settings` then `Project Settings`. This will open a JSON file which you can edit to enter your desired settings.

> This process is easiest with either a Pycom Expansion Board or a Pytrack/Pysense as the addresses are automatically selected. For external products such as FTDI USB Serial Cables, the serial address may need to be copied manually. Additionally, the reset button on the device may also need to be pressed before a connection message appears.

Connecting via Telnet

After installing the Pymakr Plugin, a device may be connected via the telnet interface. Please see the following steps:

1. Ensure that Pycom device is turned on
2. Connect the host computer to the WiFi Access Point named after your board (the SSID will be as follows e.g. `lopy-wlan-xxxx`, `wipy-wlan-xxxx`, etc.). The password is `www.pycom.io`.
3. Follow the steps as above in the "Connecting via Serial USB" section but enter `192.168.4.1` as the address.
4. The default username and password are `micro` and `python`, respectively.
5. Click `Connect` in the Pymakr pane, Pymakr will now connect via telnet.


Pymakr 0.0.1
Adds a REPL console to Atom that connects to your Pycom-board and can run and sync your code.
pycom
Uninstall
Disable
pycom/Pymakr-atom
This package added 165ms to startup time.
View on Atom.io
Report Issue
CHANGELOG
LICENSE
View Code
Settings
Device address
Either connect through USB serial using a comport (for example /dev/cu.usbserial-DQ00573Z) or an IP address for a telnet connection. Username and password are not needed for serial connections.
Default: 192.168.4.1
User name
Default: micro
Password
Default: python
Ctrl-c on connect
Stops all running programs when connecting to the board
Sync Folder
This folder will be uploaded to the pyboard when using the sync button (default "pythonproject" for testing purposes).

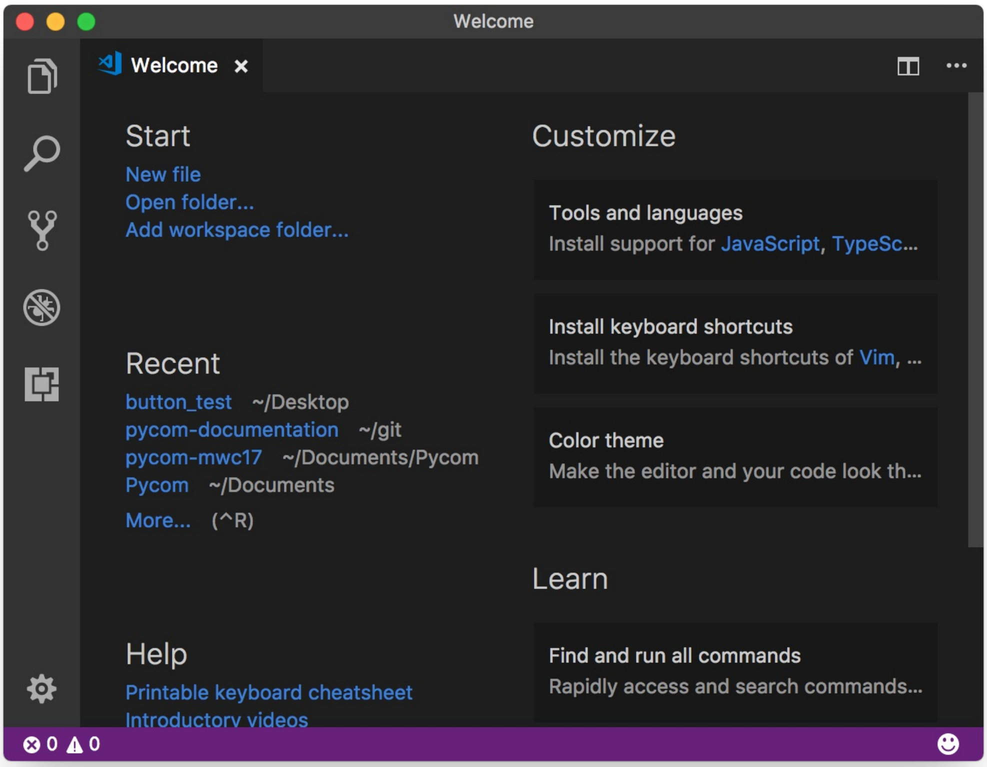
Pymakr Plugin Installation for Visual Studio Code

Pycom also supports Microsoft's Visual Studio Code IDE platform with the Pymakr Plugin. To download Visual Studio Code, navigate to VS Code.

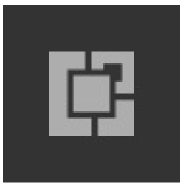
You will also need NodeJS installed on your PC. Please download the latest LTS version available from the NodeJS website.

Please follow these steps to install the Pymakr VSCode Extension:

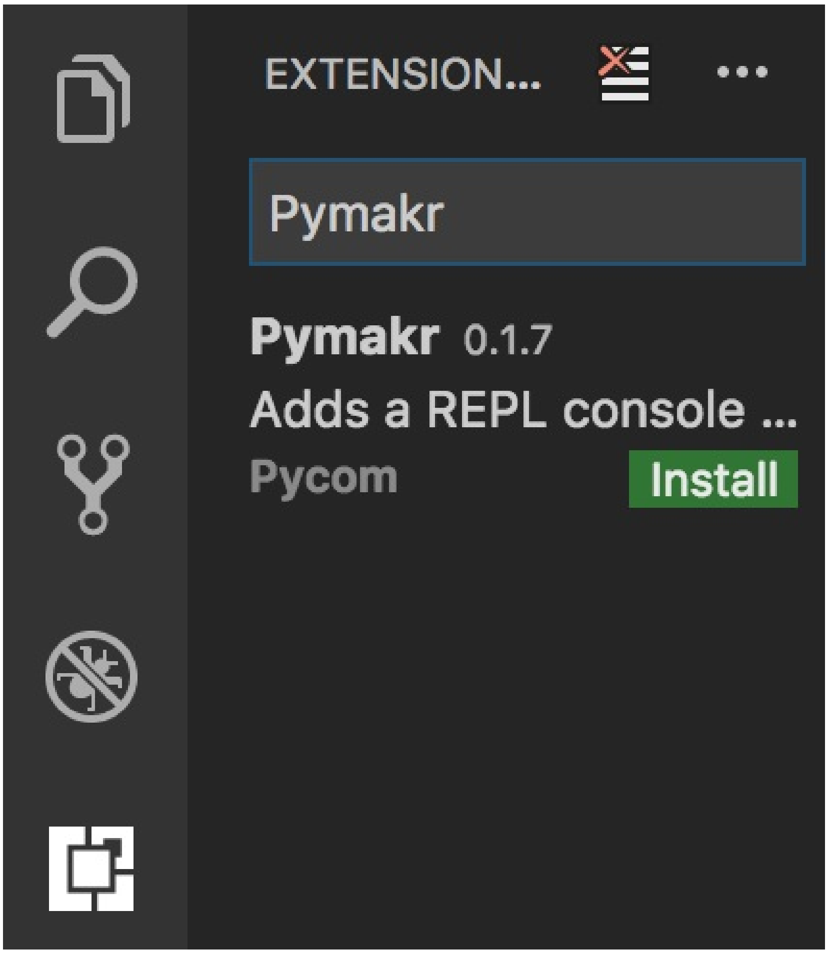
1. Ensure that you have VSCode installed and open.



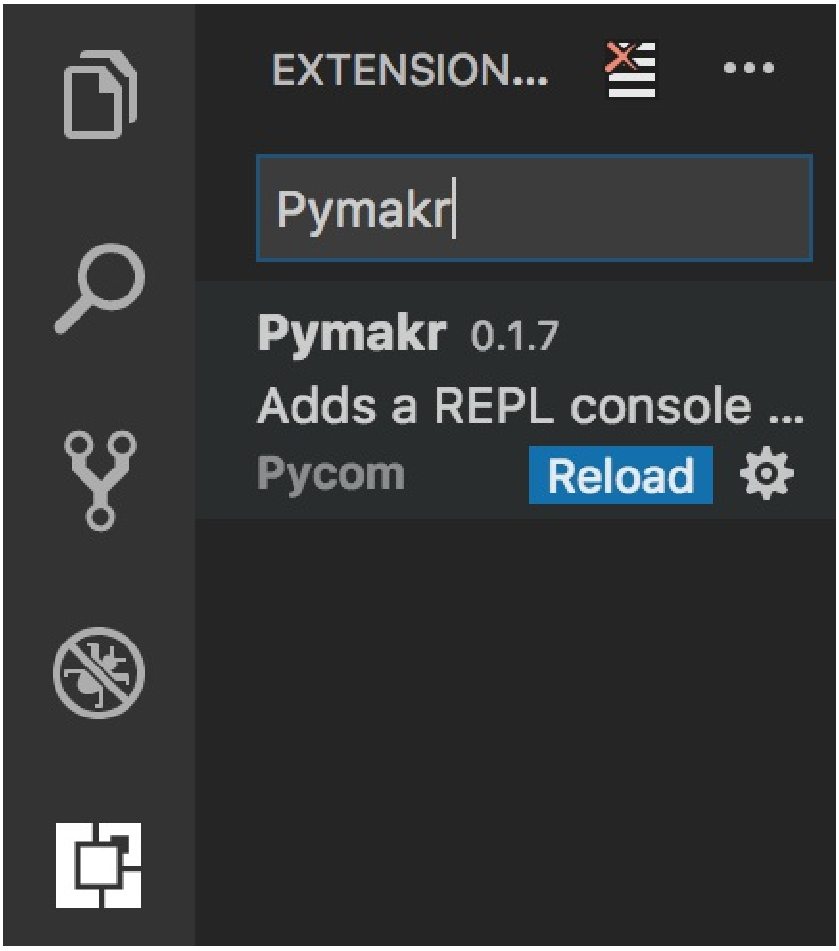
2. Navigate to the Extensions page, using the 5th button in the left navigation



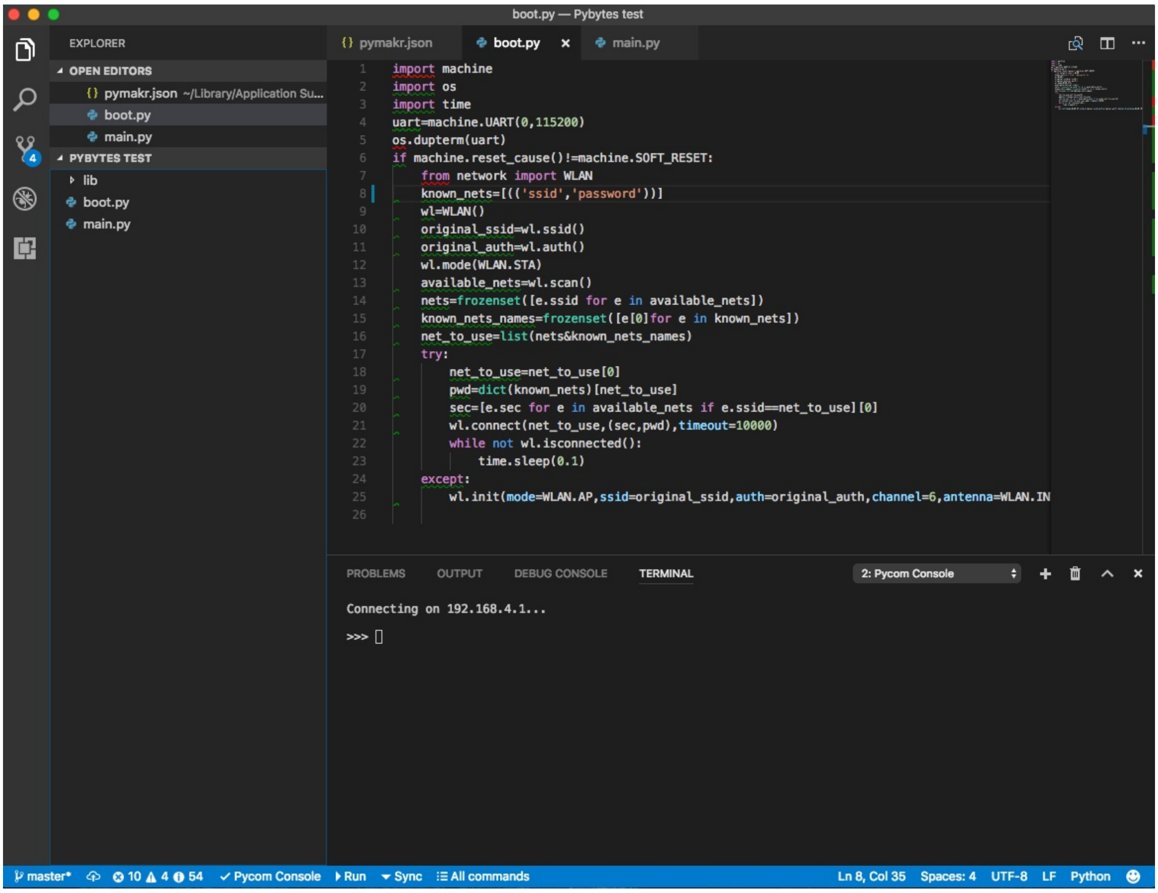
3. Search for `Pymakr` and click the install button next to it.



4. Within a few minutes, a reload button should appear. Press it to reload VSCode.

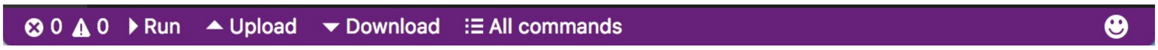


5. That's it! You've installed the Pymakr Extension for VSCode

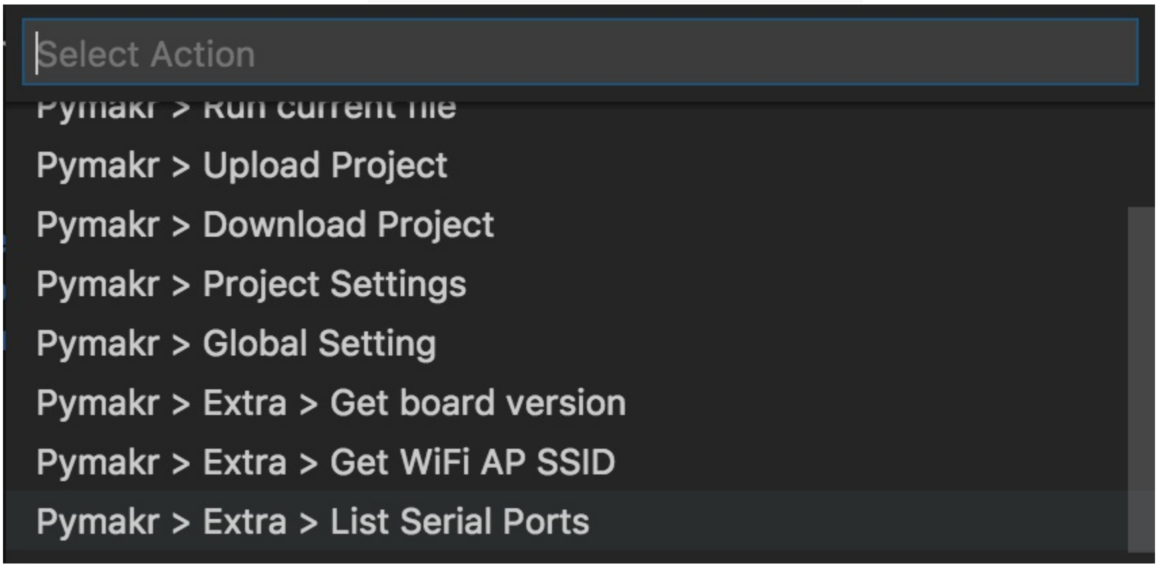


Connecting via Serial USB

After installing the Pymakr Plugin, you need to take a few seconds to configure it for first time use. Please follow these steps:

1. Connect your Pycom device to your computer via USB. If you are using an expansion board, and have just finished a firmware upgrade, be sure to **Remove the wire between GND and G23** and reset your device by pressing the button.
2. Open Visual Studio Code and ensure that the Pymakr Plugin has correctly installed.


3. Click `All commands` on the bottom of the Visual Studio Code window


4. In the list that appears, click `Pymakr > Extra > List Serial Ports`


5. This will list the available serial ports. If Pymakr is able to auto-detect which to use, this will be copied to your clipboard. If not please manually copy the correct serial port.


6. Once again click `All commands`, then click `Pymakr > Global Settings`. This will open a JSON file. Paste the serial address you copied earlier into the field `address` and save the file.

```json
{
    "address": "192.168.4.1",
    "username": "micro",
    "password": "python",
    "sync_folder": "Demos/FridgeSensor/",
    "open_on_start": true,
    "sync_file_types": "py,txt,log,json,xml",
    "ctrl_c_on_connect": false
}
```

7. Finally close the JSON file, click `All commands` , then `Pymakr > Connect` to connect your device. The Pymakr console should show three arrows `>>>` , indicating that you are connected

```
PROBLEMS    TERMINAL    ...                1: Pycom Consol
Connecting on /dev/cu.usbserial-DQ0054ES...
>>> 
```

These settings can also be applied on a per project basis by clicking `All commands` then `Pymakr > Project Settings` . This will open a JSON file which you can edit to enter your desired settings for the currently open project.

> This process is easiest with either a Pycom Expansion Board or a Pytrack/Pysense as the addresses are automatically selected. For external products such as FTDI USB Serial Cables, the serial address may need to be copied manually. Additionally, the reset button on the device may also need to be pressed before a connection message appears.

Connecting via Telnet

After installing the Pymakr Plugin, a device may be connected via the telnet interface. Please see the following steps:

1. Ensure that Pycom device is turned on
2. Connect the host computer to the WiFi Access Point named after your board (the SSID will be as follows e.g. `lopy-wlan-xxxx` , `wipy-wlan-xxxx` , etc.). The password is `www.pycom.io` .
3. Follow the steps as above in the "Connecting via Serial USB" section but enter `192.168.4.1` as the address.

4. The default username and password are `micro` and `python`, respectively.
5. Finally close the JSON file, click `All commands`, then `Pymakr > Connect`, Pymakr will now connect via telnet.

Tools and Features

Console (REPL)

MicroPython has an interactive code tool known as the REPL (Read Evaluate Print Line). The REPL allows you to run code on your device, line by line. To begin coding, go to the Pymakr Plugin Console and start typing your code. Start by making the LED change colour.

```
import pycom # we need this module to control the LED

pycom.heartbeat(False) # disable the blue blinking
pycom.rgbled(0x00ff00) # make the LED light up green in colour
```

You can change the colour by adjusting the hex RGB value.

```
pycom.rgbled(0xff0000) # now make the LED light up red in colour
```

The console can be used to run any python code, also functions or loops.

Use `print()` to output contents of variables to the console for you to read. Returned values from functions will also be displayed if they are not caught in a variable. This will not happen for code running from the main or boot files. Here you need to use `print()` to output to the console.

> Note that after writing or pasting any indented code like a function or a while loop, the user will have to press enter up to three times to tell MicroPython the code is to be closed (this is standard MicroPython & Python behaviour).
>
> Also be aware that code written into the REPL is not saved after the device is powered off/on again.

Run

To test code on a device, create a new `.py` file or open an existing one, type the desired code, save the file and then press the `Run` button. This will run the code directly onto the Pycom board and output the results of the script to the REPL.

> Changes made to files won't be automatically uploaded to the board upon restarting or exiting the `Run` feature, as the Pycom board will not store this code. In order to push the code permanently to a device, use the `Upload` feature.

Projects

Pymakr Plugin supports user projects, allowing for pre-configured settings such as default serial address/credentials, files to be ignored and folders to sync.

pymakr.conf

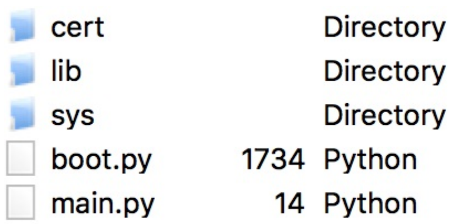
Pymakr Plugin supports local project settings using a file called `pymakr.conf`. This can be used to store the default serial address of a device, which files to ignore and other settings. An example `pymakr.conf` is shown below:

```
{
    "address": "/dev/cu.usbserial-AB001234",
    "username": "micro",
    "password": "python",
    "sync_folder": "scripts"
}
```

Upload

The Pymakr Plugins have a feature to sync and upload code to a device. This can be used for both uploading code to a device as well as testing out scripts by running them live on the device. The following steps demonstrate how to use this feature.

To start using the `Upload` feature, ensure that a project folder has been created for the device. For example, if using the `pymakr.conf` from above, this project folder should be named `scripts`. This folder should have the following structure:



Library files should be placed into the `lib` folder, certificates into the `cert` folder and so on. The `Upload` button will take the highest level folder (currently open) and upload this to the connected Pycom device. The files will be pushed to the device in exactly the same structure as within the code editor's file directory.

More

Clicking the `More` button within the Pymakr Plugin allows for some additional features. See the options below for specific functionality.

Get Firmware Version

Retrieves the firmware version of the Pycom device connected to the Pymakr Plugin instance.

Get WiFi AP SSID

Retrieves the default WiFi Access Point SSID of the Pycom device connected to the Pymakr Plugin instance.

Get Serial Ports

Retrieves the various serial ports that are available to the Pymakr Plugin instance.

Pymakr settings

Below you will find a description of the various settings available for Pymakr.

address

This is the address of the Pycom module you want Pymakr can connect to. This can be either a serial port (e.g `COM1` on windows or `/dev/cu.usbserial-DQ0054ES` on Linux/macOS) or an IP address (Telnet) (e.g. `192.168.4.1` if connected to the AP created by the Pycom module).

username

If a IP address was provided for the `address` therefore Pymakr is connecting via Telnet, you will also need to provide a username, the default is `micro` .

password

If an IP address was provided for the address, Pymakr is connecting via Telnet. You will also need to provide a password, the default is `python` .

sync_folder

If left blank, all directories inside the project will be synced to the device when the user clicks `upload` . If directories are specified, only these directories will be synced, all others will be ignored

open_on_start

If set to `true` , the Pymakr console will open and try to connect when the editor is started, or a project is opened.

safe_boot_on_upload

If set to `true` , Pymakr will reboot the connected device into safe-mode before uploading. This is useful if your code uses a lot of RAM causing issues with the upload procedure.

This feature is only available on modules running firmware version `1.17.0.b1` or higher.

sync_file_types

Only files ending with the extensions listed in this setting will be synced to the device when performing an upload. All other files are ignored. By default this is set to include: `py, txt, log, json, xml`

ctrl_c_on_connect

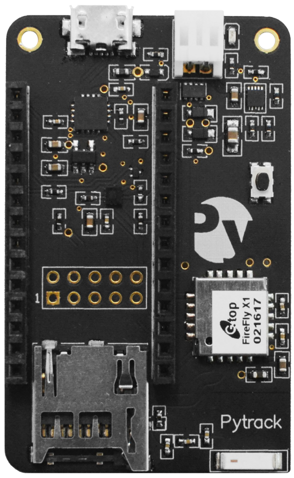
If set to `true` , Pymakr will sent the `ctrl-c` signal to the connected module before uploading. This should stop the script currently running on the device and improve the reliability of the upload process.

Pytrack & Pysense

In addition to the Expansion Board, Pycom also offers two additional sensor boards, which are ideal for quickly building a fully functioning IoT solution! Whether the application is environment sensing or asset tracking, these additional boards support a variety of sensors.

Pytrack

Pytrack is a location enabled version of the Expansion Board, intended for use in GPS applications such as asset tracking or monitoring.



Features & Hardware

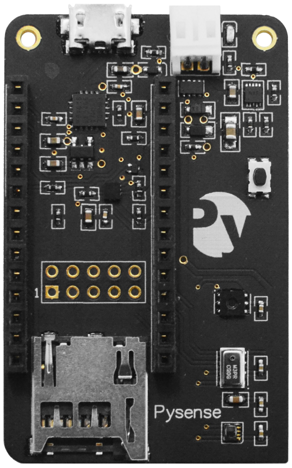
The Pytrack is has a number of features including GPS, 3-Axis Accelerometer and Battery Charger. See the list below for detailed specifics about each sensor, including datasheets.

- Serial USB
- 3-Axis Accelerometer (LIS2HH12)
- Battery Charger (BQ24040 with JST connector)
- GPS and GLONASS (L76-L)
- MicroSD Card Reader

All of the included sensors are connected to the Pycom device via the I2C interface. These pins are located at P22 (SDA) and P21 (SCL).

Pysense

Pysense is a sensor packed version of the Expansion Board, intended for use in environment sensing applications such as temperature, humidity monitoring, and light sensing.



Features & Hardware

The Pysense is packed with a number of sensors and hardware, see the list below for detailed specifics about each sensor, including datasheets.

- Serial USB
- 3-Axis Accelerometer (LIS2HH12)
- Battery Charger (BQ24040 with JST connector)
- Digital Ambient Light Sensor (LTR-329ALS-01)
- Humidity and Temperature Sensor (SI7006-A20)
- Barometric Pressure Sensor with Altimeter (MPL3115A2)
- MicroSD Card Reader

All of the included sensors are connected to the Pycom device via the I2C interface. These pins are located at `GPI09` (SDA) and `GPI08` (SCL).

Installing Software

As the development for these devices are on going with additional features being added, every week, it is essential to ensure you frequently check for updates on the Pytrack/Pysense. As well as updating the device firmware, it is important to check the GitHub repository for the respective library files as they as also being updated, to include additional features/functionality.

Updating Firmware

To update the firmware on the Pysense/Pytrack/Pyscan/Expansion Board v3, please see the following instructions. The firmware of Pysense/Pytrack/Pyscan/Expansion Board v3 can be updated via the USB port using the terminal tool, `DFU-util`.

The latest firmware DFU file can be downloaded from the links below:

- Pytrack DFU
- Pysense DFU
- Expansion Board DFU

While in the normal, application mode, the Pysense/Pytrack/Pyscan/Expansion Board v3 require a Serial USB CDC driver, in DFU, bootloader mode, the DFU driver is required. Below, the USB Product ID is depicted for each case.

Board	DFU bootloader (update mode)	Application firmware (normal mode)
Pytrack	`0xF014`	`0xF013`
Pysense	`0xF011`	`0xF012`
Pyscan	`0xEF37`	`0xEF38`
Expansion Board v3	`0xEF99`	`0xEF98`

Note: USB Vendor ID is always `0x04D8`.

Installing the DFU-util Tools

macOS

If using `homebrew`:

```
$ brew install dfu-util
```

If using `MacPorts`:

```
port install libusb dfu-util
```

Linux

Ubuntu or Debian:

```
$ sudo apt-get install dfu-util
```

Fedora:

```
$ sudo yum install dfu-util
```

Arch:

```
$ sudo pacman -Sy dfu-util
```

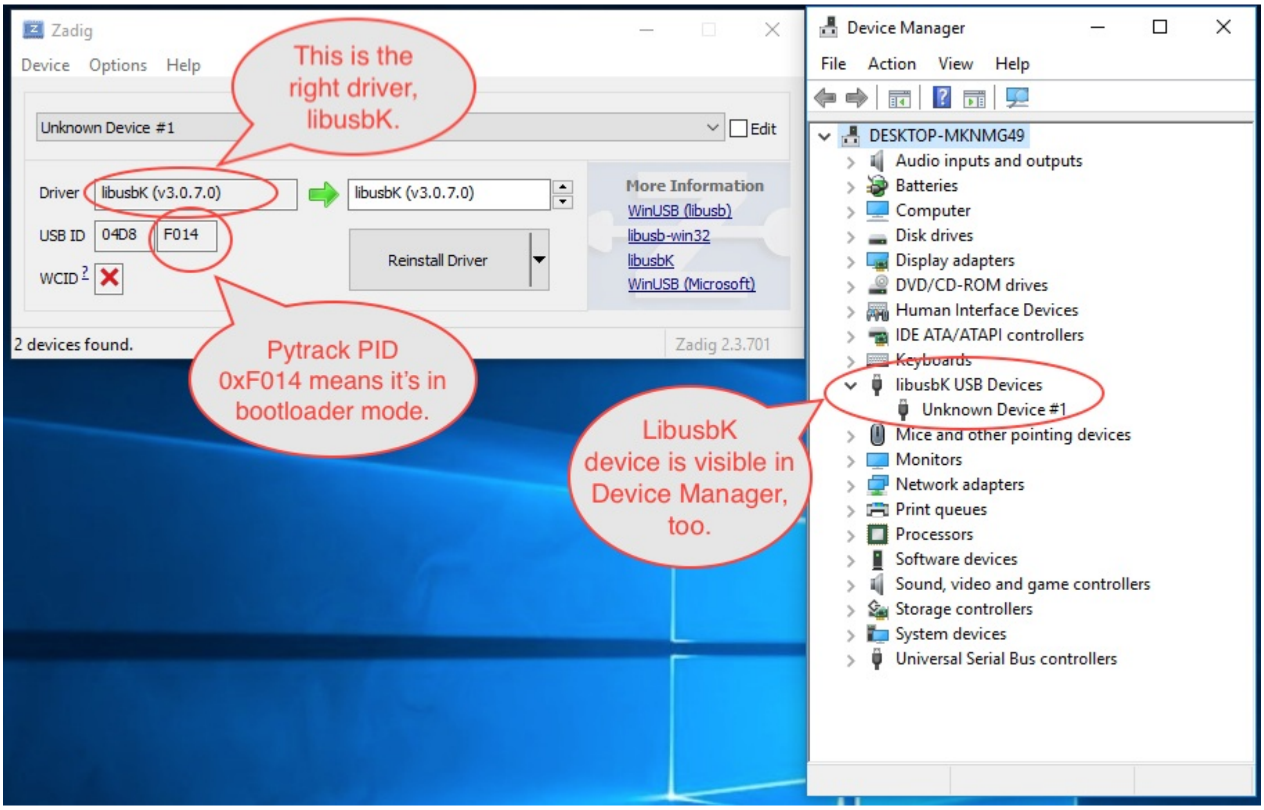
Windows

- DFU-util v0.9 – Tool to upload the firmware to the Pytrack/Pysense
- Zadig – Installer tool for the Pytrack/Pysense board DFU Firmware

To uploaded the latest DFU firmware to the Pytrack/Pysense, **first install the DFU drivers** to the host computer. Open Zadig and select `libusbK` as the driver.

To install the drivers, the Pytrack/Pysense board must be in DFU-mode:

1. Disconnect the USB cable
2. Hold down the button on the shield
3. Connect the USB cable
4. Keep the button pressed for at least one second
5. Release the button. When the board is connected in DFU-mode, it will be in this state for 7 seconds.
6. Click the `“Install Driver` button immediately. If the driver was unsuccessful, repeat from step 1.
    - *Here the USB ID has to be the DFU-bootloader one (* `0xF014` *for Pytrack or* `0xF011` *for Pysense).*
    - *This is a successful DFU driver installation for Pytrack:*

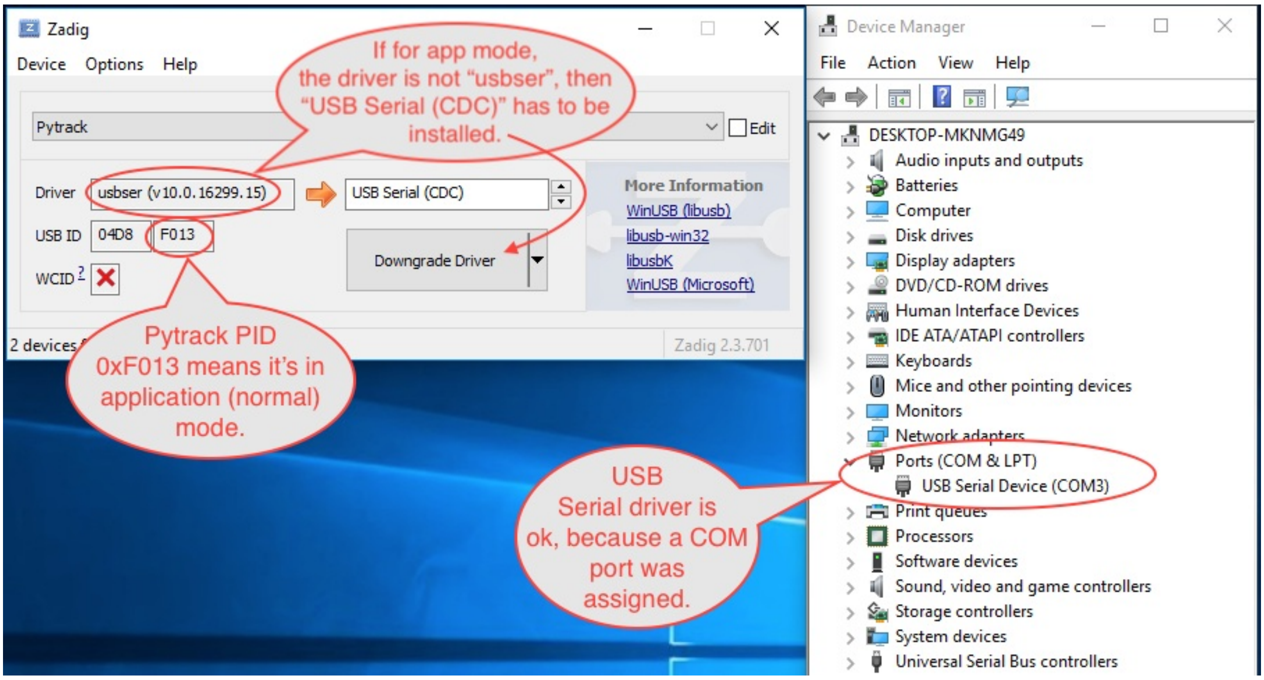


Open the command prompt and navigate to the directory where the DFU-util and the firmware was downloaded (must be in same directory). Repeat the procedure to get the board in DFU-mode and run the command below but replace `X.X.X` with the firmware version and replace Pysense with Pytrack if it is the Pytrack that is to be updated (e.g: `pytrack_0.0.8.dfu`):

```
dfu-util-static.exe -D pysense_X.X.X.dfu
```

If the update was successful, a message,"Done!" should appear in the bottom of the command prompt.

Double-check Serial USB (CDC) driver is installed in Application mode: if, by mistake, the `libusbk` driver was installed while the USB ID is the Application mode (`0xF013` for Pytrack or `0xF012` for Pysense), then the `Serial USB (CDC)` driver has to be installed for application mode. This will allow Windows to allocate a COM port, which is required for REPL console.



Using DFU-util with Pytrack, Pysense and Expansion Board v3

To enter update mode follow these steps:

1. Unplug the device
2. Press the button and keep it held (on the Expansion Board the `S1` button)
3. Plug in the USB cable to the host computer and wait 1 second before releasing the button
4. After this you will have approximately 7 seconds to run the DFU-util tool

MacOS and Linux:

```
$ dfu-util -D pytrack_0.0.8.dfu
```

An output, similar to the one below, will appear upon successful installation:

```
dfu-util 0.9

Copyright 2005-2009 Weston Schmidt, Harald Welte and OpenMoko Inc.
Copyright 2010-2016 Tormod Volden and Stefan Schmidt
This program is Free Software and has ABSOLUTELY NO WARRANTY
Please report bugs to http://sourceforge.net/p/dfu-util/tickets/

Match vendor ID from file: 04d8
Match product ID from file: f014
Opening DFU capable USB device...
ID 04d8:f014
Run-time device DFU version 0100
Claiming USB DFU Runtime Interface...
Determining device status: state = dfuIDLE, status = 0
dfu-util: WARNING: Runtime device already in DFU state ?!?
Claiming USB DFU Interface...
Setting Alternate Setting #0 ...
Determining device status: state = dfuIDLE, status = 0
dfuIDLE, continuing
DFU mode device DFU version 0100
Device returned transfer size 64
Copying data from PC to DFU device
Download    [=========================] 100%        16384 bytes
Download done.
state(2) = dfuIDLE, status(0) = No error condition is present
Done!
```

Debugging

Using `lsusb` command, the Pytrack/Pysense device should be visible in both normal and bootloader modes.

For exemple, a Pytrack board is visible as either:

- `Bus 020 Device 004: ID 04d8:f014 Microchip Technology Inc. Application Specific Device`
  - this is bootloader mode (`f014` is USB PID), active just for 7-8 seconds, if the reset button was just pressed before plugging USB connector.
- `Bus 020 Device 005: ID 04d8:f013 Microchip Technology Inc. Pytrack Serial: Pyabcde0`
  - this is normal, application mode (`f013` is USB PID), this means the bootloader verified application partition and it boot-up correctly.

Windows 7 Drivers

Pytrack and Pysense will work out of the box for Windows 8/10/+, macOS as well as Linux. If using Windows 7, drivers to support the boards will need to be installed.

Please follow the instructions below to install the required drivers.

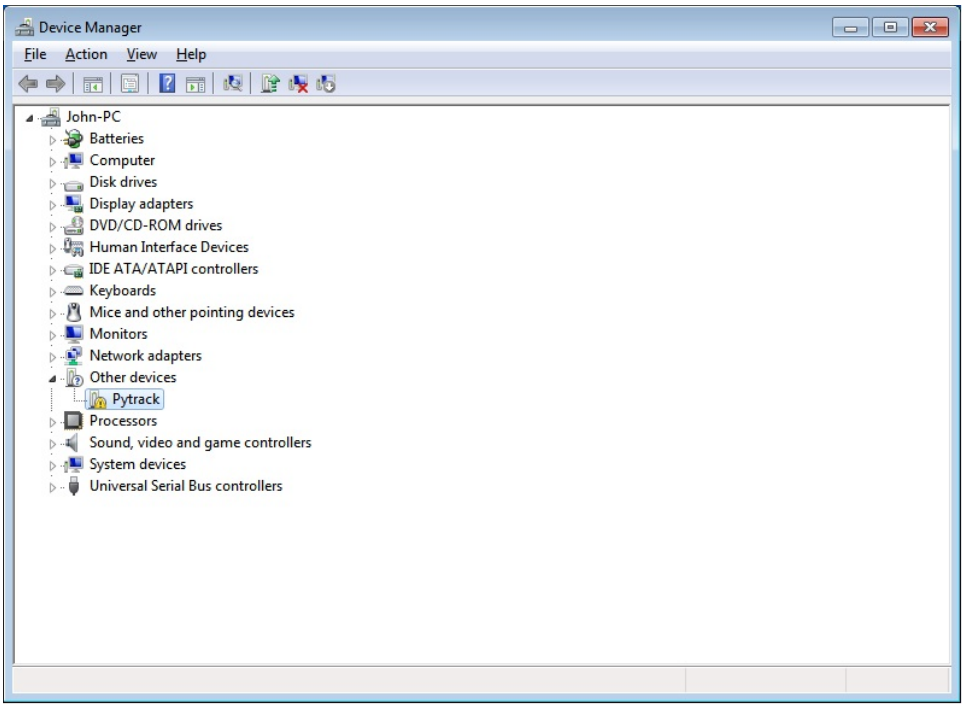
Download

Please download the driver software from the link below.

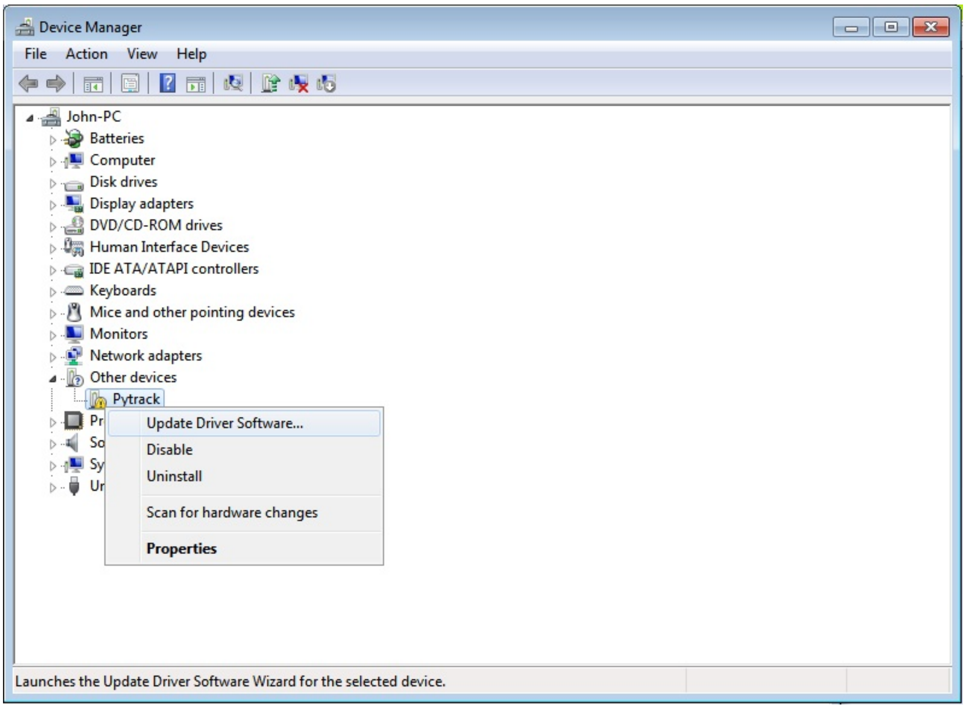
Pytrack/Pysense/Pyscan/Expansion board 3 Driver

Installation

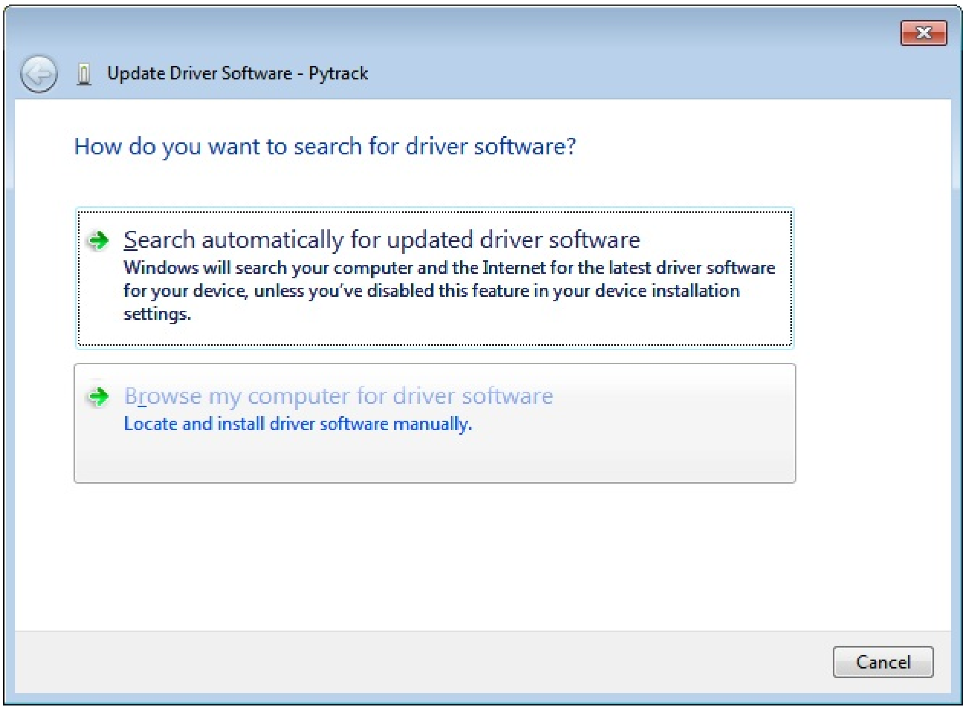
First navigate open the Windows start menu and search/navigate to `Device Manager` . You should see your Pytrack/Pysense in the dropdown under **other devices**.



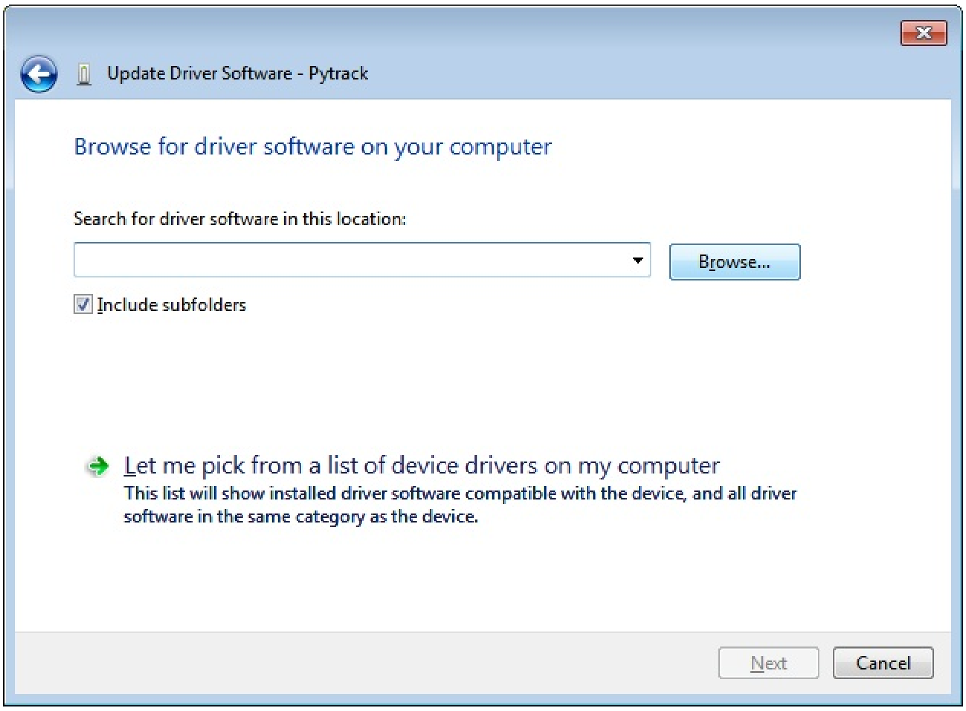
Right click the device and select `Update Driver Software` .



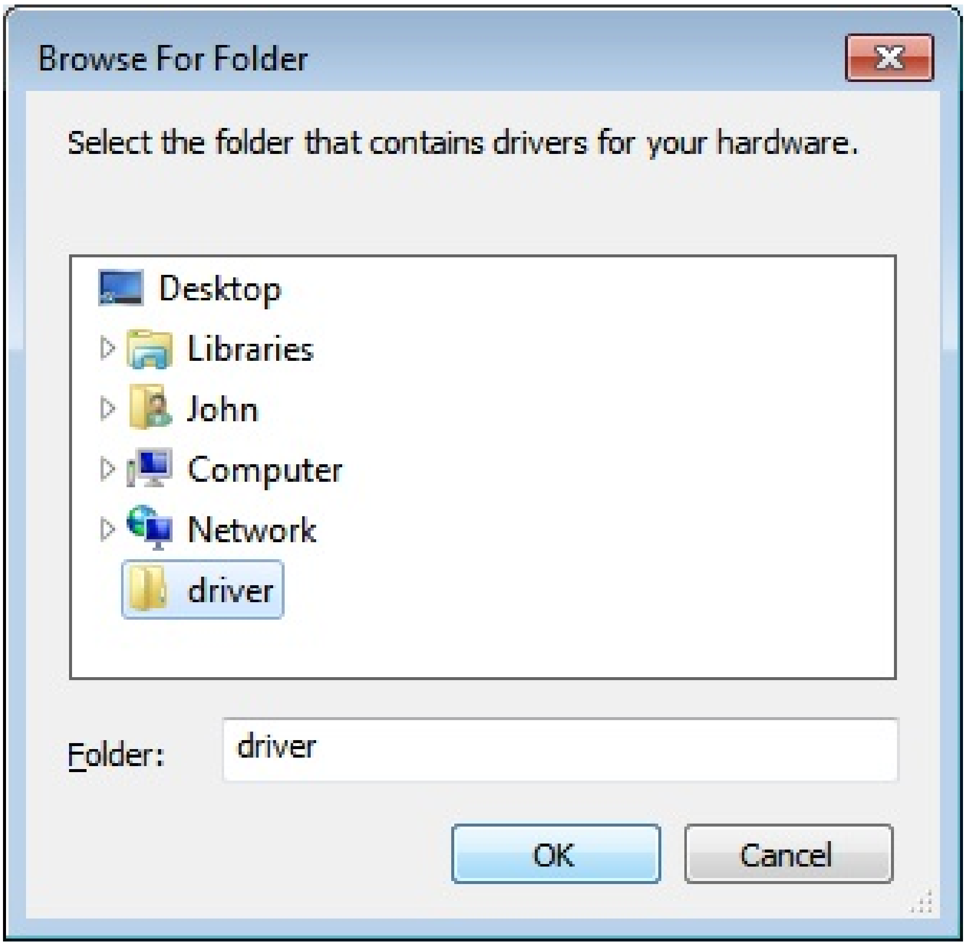
Select the option to **Browse my computer for driver software**.



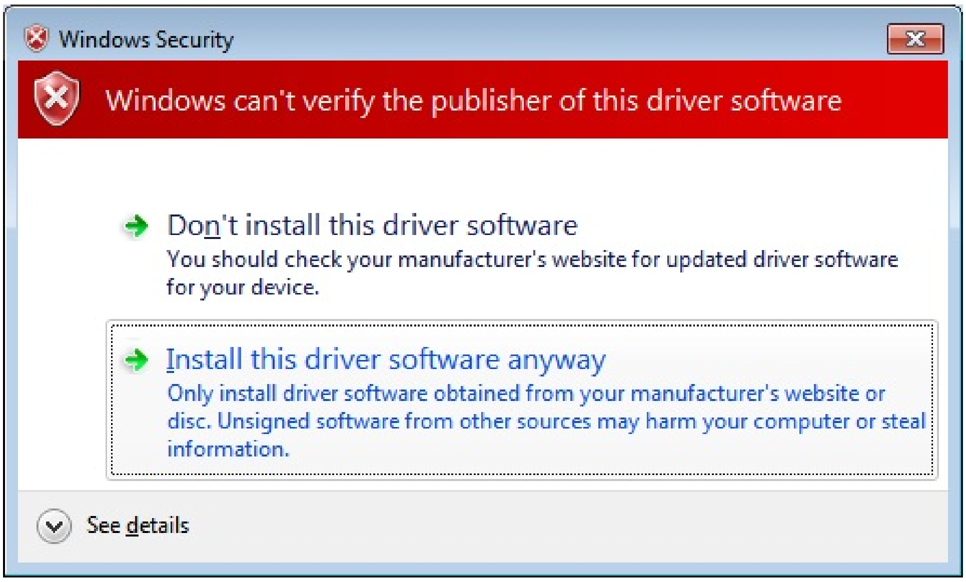
Next you will need to navigate to where you downloaded the driver to (e.g. **Downloads** Folder).



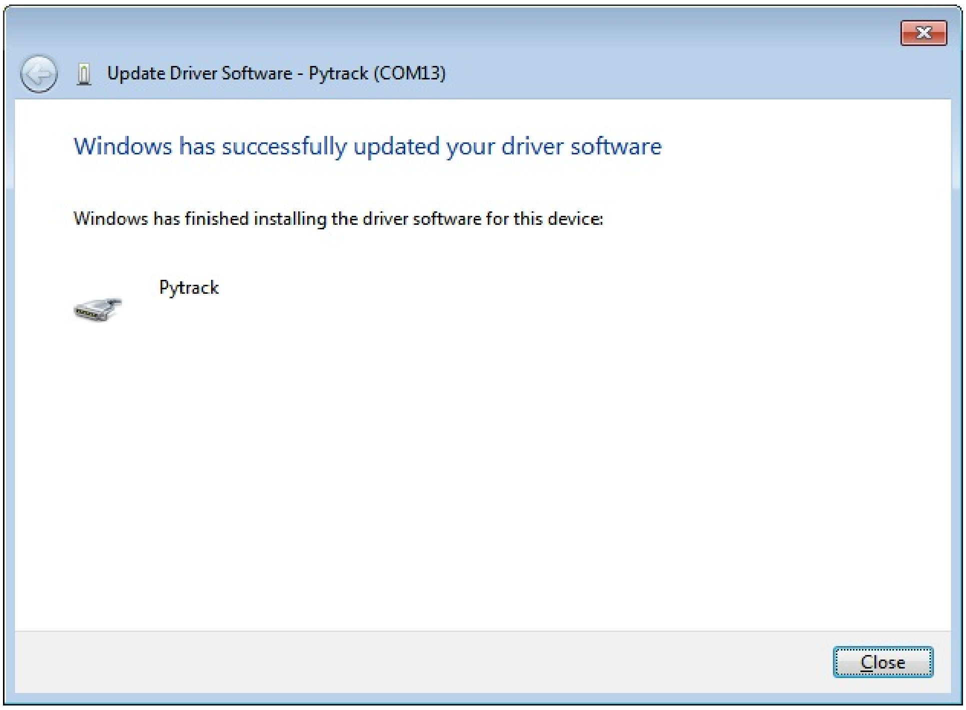
Specify the folder in which the drivers are contained. If you haven't extracted the `.zip` file, please do this before selecting the folder.



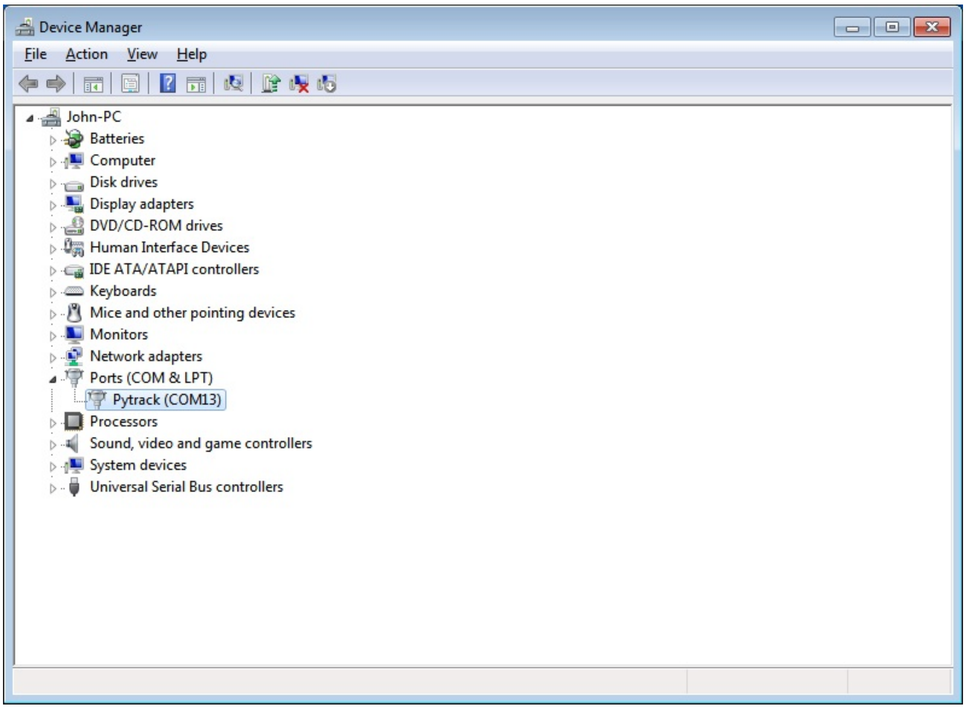
You may receive a warning, suggesting that windows can't verify the publisher of this driver. Click `Install this driver software anyway` as this link points to our official driver.



If the installation was successful, you should now see a window specifying that the driver was correctly installed.



To confirm that the installation was correct, navigate back to the `Device Manager` and click the dropdown for other devices. The warning label should now be gone and Pytrack/Pysense should be installed.


Device Manager
File
Action
View
Help
John-PC
Batteries
Computer
Disk drives
Display adapters
DVD/CD-ROM drives
Human Interface Devices
IDE ATA/ATAPI controllers
Keyboards
Mice and other pointing devices
Monitors
Network adapters
Ports (COM & LPT)
Pytrack (COM13)
Processors
Sound, video and game controllers
System devices
Universal Serial Bus controllers

Installing Libraries

To utilise the sensors on the Pytrack and Pysense, Pycom has written libraries to make reading to/from the various sensors accessible via an API. These libraries reside at the Pycom GitHub repository and the latest versions can be found under the releases page.

GitHub Repository

Download the repository as a `.zip` file, navigate to the correct device (Pysense/Pytrack), extract the files and then upload the desired files to the device in the instructions below.

Uploading the Libraries to a Device

These libraries should be uploaded to a device (LoPy, SiPy, WiPy 2.0, etc.) in the same process as a standard MicroPython library. The various `.py` files should be placed into the `/lib` folder on the device. For example, if using the Pysense and the user wishes to enable the only Accelerometer and the Light Sensor, they should place the following `.py` files into the device's `/lib` folder:

```
- pysense.py
- LIS2HH12.py
- LTR329ALS01.py
```

Add as many or as few of the libraries that are required.

In addition to the Pysense or Pytrack specific libraries, you also need to upload the **`pycoproc.py`** file from the `_lib/pycoproc_` folder inside the libraries archive.

> The Pytrack and Pysense boards behave the same as the Expansion Board. `Upload`, `Run` and upload code to Pycom modules via the Pymakr Plugin, in exactly the same process.

Importing/Using the Libraries

Once the libraries are uploaded to the device, they can be used/imported as a typical MicroPython library would be. For example, importing and using the light sensor on the Pysense:

```python
from pysense import Pysense
from LTR329ALS01 import LTR329ALS01

py = Pysense()
lt = LTR329ALS01(py)

print(lt.light())
```

API Reference

To simplify usability, APIs for the libraries have been created, abstracting away the low level interactions with the sensors. The next following pages refer to the respective libraries for the Pytrack and Pysense.

Pytrack API

This chapter describes the various libraries which are designed for the Pytrack board. This includes details about the various methods and classes available for each of the Pytrack's sensors.

3-Axis Accelerometer (LIS2HH12)

Pytrack has a 3-Axis Accelerometer that provides outputs for acceleration as well as roll, pitch and yaw.

Constructors

class LIS2HH12(pytrack = None, sda = 'P22', scl = 'P21')

Creates a `LIS2HH12` object, that will return values for acceleration, roll, pitch and yaw. Constructor must be passed a Pytrack or I2C object to successfully construct.

Methods

LIS2HH12.acceleration()

Read the acceleration from the `LIS2HH12`. Returns a **tuple** with the 3 values of acceleration (G).

LIS2HH12.roll()

Read the current roll from the `LIS2HH12`. Returns a **float** in degrees in the range -180 to 180.

LIS2HH12.pitch()

Read the current pitch from the `LIS2HH12`. Returns a **float** in degrees in the range -90 to 90. Once the board tilts beyond this range the values will repeat. This is due to a lack of yaw measurement, making it not possible to know the exact orientation of the board.

GPS with GLONASS (Quectel L76-L GNSS)

Pytrack has a GPS (with GLONASS) that provides outputs longitude/latitude, speed and other information about the Pytrack's location.

Constructors

class L76GNSS(pytrack = None, sda = 'P22', scl = 'P21', timeout = None)

Creates a `L76GNSS` object, that will return values for longitude and latitude. Constructor must be passed a Pytrack or I2C object to successfully construct. Set the `timeout` to a time period (in seconds) for the GPS to search for a lock. If a lock is not found by the time the `timeout` has expired, the `coordinates` method will return `(None, None)` .

Methods

L76GNSS.coordinates(debug = False)

Read the longitude and latitude from the `L76GNSS` . Returns a **tuple** with the longitude and latitude. With `debug` set to `True` the output from the GPS is verbose.

> Please note that more functionality is being added weekly to these libraries. If a required feature is not available, feel free to contribute with a pull request at the Libraries GitHub repository

Pysense API

This chapter describes the various libraries which are designed for the Pysense Board. This includes details about the various methods and classes available for each of the Pysense's sensors.

3-Axis Accelerometer (LIS2HH12)

Pysense has a 3-Axis Accelerometer that provides outputs for acceleration as well as roll, pitch and yaw.

Constructors

class **LIS2HH12(pysense = None, sda = 'P22', scl = 'P21')**

Creates a `LIS2HH12` object, that will return values for acceleration, roll, pitch and yaw. Constructor must be passed a Pysense or I2C object to successfully construct.

Methods

LIS2HH12.acceleration()

Read the acceleration from the `LIS2HH12`. Returns a **tuple** with the 3 values of acceleration (G).

LIS2HH12.roll()

Read the current roll from the `LIS2HH12`. Returns a **float** in degrees in the range -180 to 180.

LIS2HH12.pitch()

Read the current pitch from the `LIS2HH12`. Returns a **float** in degrees in the range -90 to 90. Once the board tilts beyond this range the values will repeat. This is due to a lack of yaw measurement, making it not possible to know the exact orientation of the board.

Digital Ambient Light Sensor (LTR-329ALS-01)

Pysense has a dual light sensor that provides outputs for external light levels in lux. See the datasheet for more information about the wavelengths of the two sensors.

Constructors

***class* LTR329ALS01(pysense = None, sda = 'P22', scl = 'P21', gain = ALS_GAIN_1X, integration = ALS_INT_100, rate = ALS_RATE_500)**

Creates a `LTR329ALS01` object, that will return values for light in lux. Constructor must be passed a Pysense or I2C object to successfully construct.

Methods

LTR329ALS01.light()

Read the light levels of both `LTR329ALS01` sensors. Returns a **tuple** with two values for light levels in lux.

Arguments

The following arguments may be passed into the constructor.

gain

ALS_GAIN_1X, ALS_GAIN_2X, ALS_GAIN_4X, ALS_GAIN_8X, ALS_GAIN_48X, ALS_GAIN_96X
integration

ALS_INT_50, ALS_INT_100, ALS_INT_150, ALS_INT_200, ALS_INT_250, ALS_INT_300, ALS_INT_350, ALS_INT_400
rate

ALS_RATE_50, ALS_RATE_100, ALS_RATE_200, ALS_RATE_500, ALS_RATE_1000, ALS_RATE_2000

Humidity and Temperature Sensor (SI7006A20)

Pysense has a Humidity and Temperature sensor that provides values of relative humidity and external temperature.

Constructors

***class* SI7006A20(pysense = None, sda = 'P22', scl = 'P21')**

Creates a `SI7006A20` object, that will return values for humidity (%) and temperature ('C). Constructor must be passed a Pysense or I2C object to successfully construct.

Methods

SI7006A20.humidity()

Read the relative humidity of the `SI7006A20`. Returns a **float** with the percentage relative humidity.

SI7006A20.temperature()

Read the external temperature of the `SI7006A20`. Returns a **float** with the temperature.

Barometric Pressure Sensor with Altimeter (MPL3115A2)

Pysense has a Barometric Pressure sensor that provides readings for pressure, altitude as well as an additional temperature sensor.

Constructors

class MPL3115A2(pysense = None, sda = 'P22', scl = 'P21', mode = PRESSURE)

Creates a `MPL3115A2` object, that will return values for pressure (Pa), altitude (m) and temperature ('C). Constructor must be passed a Pysense or I2C object to successfully construct.

Methods

MPL3115A2.pressure()

Read the atmospheric pressure of the `MPL3115A2`. Returns a **float** with the pressure in (Pa).

MPL3115A2.altitude()

Read the altitude of the `MPL3115A2`. Returns a **float** with the altitude in (m).

MPL3115A2.temperature()

Read the temperature of the `MPL3115A2`. Returns a **float** with the temperature in ('C).

Arguments

The following arguments may be passed into the constructor.

mode

PRESSURE, ALTITUDE

> Please note that more functionality is being added weekly to these libraries. If a required feature is not available, feel free to contribute with a pull request at the Libraries GitHub repository

Sleep and Wakeup for Pytrack/Pysense API

This chapter describes the various methods for sleep and wakeup which are embedded in Pytrack and Pysense libraries. Both Pytrack and Pysense have the same methods, although the appropriate class, either `pytrack` or `pysense`, has to be instantiated.

Quick Usage Example

The following example is also available at Sleep Wakeup Example Libraries GitHub repository

```
#from pytrack import Pytrack
from pysense import Pysense
from LIS2HH12 import LIS2HH12
import time

#py = Pytrack()
py = Pysense()

# display the reset reason code and the sleep remaining in seconds
# possible values of wakeup reason are:
# WAKE_REASON_ACCELEROMETER = 1
# WAKE_REASON_PUSH_BUTTON = 2
# WAKE_REASON_TIMER = 4
# WAKE_REASON_INT_PIN = 8

print("Wakeup reason: " + str(py.get_wake_reason()))
print("Approximate sleep remaining: " + str(py.get_sleep_remaining()) + " sec")
time.sleep(0.5)

# enable wakeup source from INT pin
py.setup_int_pin_wake_up(False)

acc = LIS2HH12()

# enable activity and also inactivity interrupts, using the default callback handler
py.setup_int_wake_up(True, True)

# set the acceleration threshold to 2000mG (2G) and the min duration to 200ms
acc.enable_activity_interrupt(2000, 200)

# go to sleep for 5 minutes maximum if no accelerometer interrupt happens
py.setup_sleep(300)
py.go_to_sleep()
```

Methods

pytrack.get_sleep_remaining()

In the event of a sleep session that was awoken by an asynchronous event (Accelerometer, INT pin or Reset button) the approximate sleep remaining interval (expressed in **seconds**) can be found out. The user has to manually use `setup_sleep()` to configure the next sleep interval.

pytrack.get_wake_reason()

Returns the last wakeup reason. Possible values are:

```
# WAKE_REASON_ACCELEROMETER = 1 # Accelerometer activity/inactivity detection
# WAKE_REASON_PUSH_BUTTON = 2   # Pytrack/Pysense reset buttom
# WAKE_REASON_TIMER = 4         # Normal timeout of the sleep interval
# WAKE_REASON_INT_PIN = 8       # INT pin
```

Note: the `WAKE_REASON_INT_PIN` can be used if the PIC_RC1 pin (pin#6 on External IO Header) is toggled.

As in the above example, this method should be called at the beginning of the script, to find out the reset (wakeup) reason.

pytrack.go_to_sleep([gps=True])

Puts the board in sleep mode, for the duration, which has to be set previously with `pytrack.setup_sleep(timout_sec)`. The optional boolean parameter sets the GPS state during sleep.

MicroPython code, which is after this function, is not executed, as wakeup will restart MicroPython.

pytrack.setup_int_wake_up(rising, falling])

Enables as wakeup source, the accelerometer INT pin (PIC - RA5). The boolean parameters will indicate rising edge (activity detection) and/or falling edge (inactivity detection) is configured.

The accelerometer (class `LIS2HH12`) has to be also configured for a certain acceleration threshold and duration. Code snippet:

```
from pytrack import Pytrack
from LIS2HH12 import LIS2HH12

py = Pytrack()
acc = LIS2HH12()

# enable activity and also inactivity interrupts, using the default callback handler
py.setup_int_wake_up(True, True)

# set the acceleration threshold to 2000mG (2G) and the min duration to 200ms
acc.enable_activity_interrupt(2000, 200)
```

pytrack.setup_int_pin_wake_up([rising_edge = True])

Enables as wakeup source, the INT pic (PIC - RC1, pin#6 on External IO Header). Either rising or falling edge has to be set, by default it's rising edge.

pytrack.setup_sleep(time_seconds)

Sets the sleep interval, specified in seconds. The actual sleep will be started by calling `go_to_sleep()` method.

Please note that more functionality is being added weekly to these libraries. If a required feature is not available, feel free to contribute with a pull request at the Libraries GitHub repository

Tutorials and Examples

This section contains tutorials and examples for use with Pycom modules and Expansion boards.

General Pycom tutorials contains tutorials that may be run on any Pycom device, such as connecting to a WiFi network, Bluetooth, controlling I/O pins etc. Later sections are specific to the LoPy and SiPy devices such as setting up a LoRa node or connecting to the Sigfox network. The final sections are related to examples using the Pytrack and Pysense.

Before starting, ensure that any Pycom devices are running the latest firmware; for instructions see Firmware Updates.

The source code for these tutorials, along with the required libraries can be found in in the pycom-libraries repository.

All Pycom Device Examples

This section contains generic examples that will work across all Pycom devices and Expansion Boards.

Using the REPL Prompt

Using the Pymakr Plugin, open and connect a device or use serial terminal (PuTTY, screen, picocom, etc). Upon connecting, there should be a blank screen with a flashing cursor. Press Enter and a MicroPython prompt should appear, i.e. `>>>`. Let's make sure it is working with the obligatory test:

```
>>> print("Hello LoPy!")
Hello LoPy!
```

In the example above, the `>>>` characters should not be typed. They are there to indicate that the text should be placed after the prompt. Once the text has been entered `print("Hello LoPy!")` and pressed `Enter`, the output should appear on screen, identical to the example above.

Basic Python commands can be tested out in a similar fashion.

If this is not working, try either a hard reset or a soft reset; see below.

Here are some other example, utilising the device's hardware features:

```
>>> from machine import Pin
>>> led = Pin('G16', mode=Pin.OUT, value=1)
>>> led(0)
>>> led(1)
>>> led.toggle()
>>> 1 + 2
3
>>> 5 / 2
2.5
>>> 20 * 'py'
'pypypypypypypypypypypypypypypypypypypypy'
```

Resetting the Device

If something goes wrong, the device can be reset with two methods. The first is to press `CTRL-D` at the MicroPython prompt, which will perform a soft reset. A message, as following, will appear:

```
>>>
PYB: soft reboot
MicroPython v1.4.6-146-g1d8b5e5 on 2016-10-21; LoPy with ESP32
Type "help()" for more information.
>>>
```

If that still isn't working a hard reset can be performed (power-off/on) by pressing the `RST` switch (the small black button next to the RGB LED). Using telnet, this will end the session, disconnecting the program that was used to connect to the Pycom Device.

WLAN

The WLAN is a system feature of all Pycom devices, therefore it is enabled by default.

In order to retrieve the current WLAN instance, run:

```
>>> from network import WLAN
>>> wlan = WLAN() # we call the constructor without params
```

The current mode (`WLAN.AP` after power up) may be checked by running:

```
>>> wlan.mode()
```

When changing the WLAN mode, if following the instructions below, the WLAN connection to the Pycom device will be broken. This means commands will not run interactively over WiFi.

There are two ways around this:

1. Put this setup code into the `boot.py` file of the Pycom device so that it gets executed automatically after reset.
2. Duplicate the REPL on UART. This way commands can be run via Serial USB.

Connecting to a Router

The WLAN network class always boots in `WLAN.AP` mode; to connect it to an existing network, the WiFi class must be configured as a station:

```
from network import WLAN
wlan = WLAN(mode=WLAN.STA)
```

Now the device may proceed to scan for networks:

```
nets = wlan.scan()
for net in nets:
    if net.ssid == 'mywifi':
        print('Network found!')
        wlan.connect(net.ssid, auth=(net.sec, 'mywifikey'), timeout=5000)
        while not wlan.isconnected():
            machine.idle() # save power while waiting
        print('WLAN connection succeeded!')
        break
```

Assigning a Static IP Address at Boot Up

If the users wants their device to connect to a home router upon boot up, using with a fixed IP address, use the following script as `/flash/boot.py` :

```
import machine
from network import WLAN
wlan = WLAN() # get current object, without changing the mode

if machine.reset_cause() != machine.SOFT_RESET:
    wlan.init(mode=WLAN.STA)
    # configuration below MUST match your home router settings!!
    wlan.ifconfig(config=('192.168.178.107', '255.255.255.0', '192.168.178.1', '8.8.8.
8'))

if not wlan.isconnected():
    # change the line below to match your network ssid, security and password
    wlan.connect('mywifi', auth=(WLAN.WPA2, 'mywifikey'), timeout=5000)
    while not wlan.isconnected():
        machine.idle() # save power while waiting
```

Notice how we check for the reset cause and the connection status, this is crucial in order to be able to soft reset the LoPy during a telnet session without breaking the connection.

Multiple Networks using a Static IP Address

The following script holds a list with nets and an optional list of `wlan_config` to set a fixed IP

```
import os
import machine

uart = machine.UART(0, 115200)
os.dupterm(uart)

known_nets = {
    '<net>': {'pwd': '<password>'},
    '<net>': {'pwd': '<password>', 'wlan_config':  ('10.0.0.114', '255.255.0.0', '10.0
.0.1', '10.0.0.1')}, # (ip, subnet_mask, gateway, DNS_server)
}

if machine.reset_cause() != machine.SOFT_RESET:
    from network import WLAN
    wl = WLAN()
    wl.mode(WLAN.STA)
    original_ssid = wl.ssid()
    original_auth = wl.auth()

    print("Scanning for known wifi nets")
    available_nets = wl.scan()
    nets = frozenset([e.ssid for e in available_nets])

    known_nets_names = frozenset([key for key in known_nets])
    net_to_use = list(nets & known_nets_names)
    try:
        net_to_use = net_to_use[0]
        net_properties = known_nets[net_to_use]
        pwd = net_properties['pwd']
        sec = [e.sec for e in available_nets if e.ssid == net_to_use][0]
        if 'wlan_config' in net_properties:
            wl.ifconfig(config=net_properties['wlan_config'])
        wl.connect(net_to_use, (sec, pwd), timeout=10000)
        while not wl.isconnected():
            machine.idle() # save power while waiting
        print("Connected to "+net_to_use+" with IP address:" + wl.ifconfig()[0])

    except Exception as e:
        print("Failed to connect to any known network, going into AP mode")
        wl.init(mode=WLAN.AP, ssid=original_ssid, auth=original_auth, channel=6, anten
na=WLAN.INT_ANT)
```

Connecting to a WPA2-Enterprise network

Connecting with EAP-TLS:

Before connecting, obtain and copy the public and private keys to the device, e.g. under location `/flash/cert`. If it is required to validate the server’s public key, an appropriate CA certificate (chain) must also be provided.

```
from network import WLAN

wlan = WLAN(mode=WLAN.STA)
wlan.connect(ssid='mywifi', auth=(WLAN.WPA2_ENT,), identity='myidentity', ca_certs='/f
lash/cert/ca.pem', keyfile='/flash/cert/client.key', certfile='/flash/cert/client.crt'
)
```

Connecting with EAP-PEAP or EAP-TTLS:

In case of EAP-PEAP (or EAP-TTLS), the client key and certificate are not necessary, only a username and password pair. If it is required to validate the server's public key, an appropriate CA certificate (chain) must also be provided.

```
from network import WLAN

wlan = WLAN(mode=WLAN.STA)
wlan.connect(ssid='mywifi', auth=(WLAN.WPA2_ENT, 'username', 'password'), identity='my
identity', ca_certs='/flash/cert/ca.pem')
```

Bluetooth

At present, basic BLE functionality is available. More features will be implemented in the near future, such as pairing. This page will be updated in line with these features.

Full info on `bluetooth` can be found within Bluetooth page of the Firmware API Reference.

Scan for BLE Devices

Scan for all of the advertising devices within range of the scanning device.

```
bluetooth.start_scan(10)  # starts scanning and stop after 10 seconds
bluetooth.start_scan(-1)  # starts scanning indefinitely until bluetooth.stop_scan() i
s called
```

Raw Data from a BLE Device

A quick usage example that scans and prints the raw data from advertisements.

```
from network import Bluetooth

bluetooth = Bluetooth()
bluetooth.start_scan(-1)    # start scanning with no timeout

while True:
    print(bluetooth.get_adv())
```

Connect to a BLE Device

Connecting to a device that is sending advertisements.

```
from network import Bluetooth
import ubinascii
bluetooth = Bluetooth()

# scan until we can connect to any BLE device around
bluetooth.start_scan(-1)
adv = None
while True:
    adv = bluetooth.get_adv()
    if adv:
        try:
            bluetooth.connect(adv.mac)
        except:
            # start scanning again
            bluetooth.start_scan(-1)
            continue
        break
print("Connected to device with addr = {}".format(ubinascii.hexlify(adv.mac)))
```

Connect to a BLE Device and Retrieve Data

Connecting to a device named 'Heart Rate' and receiving data from it’s services.

```python
from network import Bluetooth
import time
bt = Bluetooth()
bt.start_scan(-1)

while True:
  adv = bt.get_adv()
  if adv and bt.resolve_adv_data(adv.data, Bluetooth.ADV_NAME_CMPL) == 'Heart Rate':
      try:
          conn = bt.connect(adv.mac)
          services = conn.services()
          for service in services:
              time.sleep(0.050)
              if type(service.uuid()) == bytes:
                  print('Reading chars from service = {}'.format(service.uuid()))
              else:
                  print('Reading chars from service = %x' % service.uuid())
              chars = service.characteristics()
              for char in chars:
                  if (char.properties() & Bluetooth.PROP_READ):
                      print('char {} value = {}'.format(char.uuid(), char.read()))
          conn.disconnect()
          break
      except:
          pass
  else:
      time.sleep(0.050)
```

Retrieve the Name & Manufacturer from a BLE Device

Using `resolve_adv_data()` to attempt to retrieve the name and manufacturer data from the advertiser.

```
import ubinascii
from network import Bluetooth
bluetooth = Bluetooth()

bluetooth.start_scan(20)
while bluetooth.isscanning():
    adv = bluetooth.get_adv()
    if adv:
        # try to get the complete name
        print(bluetooth.resolve_adv_data(adv.data, Bluetooth.ADV_NAME_CMPL))

        mfg_data = bluetooth.resolve_adv_data(adv.data, Bluetooth.ADV_MANUFACTURER_DAT
A)

        if mfg_data:
            # try to get the manufacturer data (Apple's iBeacon data is sent here)
            print(ubinascii.hexlify(mfg_data))
```

HTTPS

Basic connection using `ssl.wrap_socket()` .

```
import socket
import ssl

s = socket.socket()
ss = ssl.wrap_socket(s)
ss.connect(socket.getaddrinfo('www.google.com', 443)[0][-1])
```

Below is an example using certificates with the blynk cloud.

Certificate was downloaded from the blynk examples folder and placed in `/flash/cert/` on the device.

```
import socket
import ssl

s = socket.socket()
ss = ssl.wrap_socket(s, cert_reqs=ssl.CERT_REQUIRED, ca_certs='/flash/cert/ca.pem')
ss.connect(socket.getaddrinfo('cloud.blynk.cc', 8441)[0][-1])
```

For more info, check the `ssl` module in the API reference.

MQTT

MQTT is a lightweight messaging protocol that is ideal for sending small packets of data to and from IoT devices via WiFi.

The broker used in this example is the IO Adafruit) platform, which is free and allows for tinkering with MQTT.

Visit IO Adafruit and create an account. You'll need to get hold of an API Key as well as your credentials. Visit this guide for more information about MQTT and how to use it with Adafruit's Broker.

This example will send a message to a topic on the Adafruit MQTT broker and then also subscribe to the same topic, in order to show how to use the subscribe functionality.

```
from mqtt import MQTTClient
from network import WLAN
import machine
import time

def sub_cb(topic, msg):
   print(msg)

wlan = WLAN(mode=WLAN.STA)
wlan.connect("yourwifinetwork", auth=(WLAN.WPA2, "wifipassword"), timeout=5000)

while not wlan.isconnected():
    machine.idle()
print("Connected to WiFi\n")

client = MQTTClient("device_id", "io.adafruit.com",user="your_username", password="you
r_api_key", port=1883)

client.set_callback(sub_cb)
client.connect()
client.subscribe(topic="youraccount/feeds/lights")

while True:
    print("Sending ON")
    client.publish(topic="youraccount/feeds/lights", msg="ON")
    time.sleep(1)
    print("Sending OFF")
    client.publish(topic="youraccount/feeds/lights", msg="OFF")
    client.check_msg()

    time.sleep(1)
```

Amazon Web Services

The AWS IoT platform enables devices to connect to the Amazon cloud and lets applications in the cloud interact with Internet-connected things. Common IoT applications either collect and process telemetry from devices or enable users to control a device remotely. Things report their state by publishing messages, in JSON format, on MQTT topics.

For more information see this PDF File.

Getting Started with AWS IoT

Creating the message broker (Amazon website):

- Sign in to the AWS Management Console
- Navigate to the IoT Console by clicking on the AWS IoT link
- In the left navigation pane, choose Register/Manage
- Click on the create button, give your device a name and press create
- Click on the device that has been created
- On the Details page, in the left navigation pane, choose Security
- On the Certificates page, choose Create certificate
- Download all the certificates, then press the Activate and the Attach a Policy buttons. See image
- Click on the Create New Policy button
- On the Create Policy page, choose a policy name and the actions to authorise.
- Go to the certificates page, click on the three dots of your certificate and attach the policy to the certificate as shown in the diagram

Setting up the device (Pycom device):

- Download the latest sample code from the Pycom GitHub Repository.
- Connect to the device via FTP and put the root CA certificate, the client certificate (`*.pem.crt`) and the private key (`*.private.pem.key`) in the `/flash/cert` folder.
- Update the config file with your WiFi settings, the AWS Host and the certificate paths.
- Put the `config.py` and the `main.py` in the device flash

Configuration (`config.py`):

This file contains the WiFi, certificate paths and application specific settings that need to be updated by the user.

```
# WiFi configuration
WIFI_SSID = 'my_wifi_ssid'
WIFI_PASS = 'my_wifi_password'

# AWS general configuration
AWS_PORT = 8883
AWS_HOST = 'aws_host_url'
AWS_ROOT_CA = '/flash/cert/aws_root.ca'
AWS_CLIENT_CERT = '/flash/cert/aws_client.cert'
AWS_PRIVATE_KEY = '/flash/cert/aws_private.key'

################## Subscribe / Publish client #################
CLIENT_ID = 'PycomPublishClient'
TOPIC = 'PublishTopic'
OFFLINE_QUEUE_SIZE = -1
DRAINING_FREQ = 2
CONN_DISCONN_TIMEOUT = 10
MQTT_OPER_TIMEOUT = 5
LAST_WILL_TOPIC = 'PublishTopic'
LAST_WILL_MSG = 'To All: Last will message'

####################### Shadow updater #######################
#THING_NAME = "my thing name"
#CLIENT_ID = "ShadowUpdater"
#CONN_DISCONN_TIMEOUT = 10
#MQTT_OPER_TIMEOUT = 5

####################### Delta Listener #######################
#THING_NAME = "my thing name"
#CLIENT_ID = "DeltaListener"
#CONN_DISCONN_TIMEOUT = 10
#MQTT_OPER_TIMEOUT = 5

####################### Shadow Echo #######################
#THING_NAME = "my thing name"
#CLIENT_ID = "ShadowEcho"
#CONN_DISCONN_TIMEOUT = 10
#MQTT_OPER_TIMEOUT = 5
```

Subscibe / Publish (`main.py`)

To subscribe to a topic:

- Go to the AWS Iot page, click on manage and choose your device
- From the left hand side, choose Activity and then click MQTT client.
- Choose the topic name you entered in the configuration file.

- Messages should be published as shown in the diagram

```
# user specified callback function
def customCallback(client, userdata, message):
    print("Received a new message: ")
    print(message.payload)
    print("from topic: ")
    print(message.topic)
    print("--------------\n\n")

# configure the MQTT client
pycomAwsMQTTClient = AWSIoTMQTTClient(config.CLIENT_ID)
pycomAwsMQTTClient.configureEndpoint(config.AWS_HOST, config.AWS_PORT)
pycomAwsMQTTClient.configureCredentials(config.AWS_ROOT_CA, config.AWS_PRIVATE_KEY, co
nfig.AWS_CLIENT_CERT)

pycomAwsMQTTClient.configureOfflinePublishQueueing(config.OFFLINE_QUEUE_SIZE)
pycomAwsMQTTClient.configureDrainingFrequency(config.DRAINING_FREQ)
pycomAwsMQTTClient.configureConnectDisconnectTimeout(config.CONN_DISCONN_TIMEOUT)
pycomAwsMQTTClient.configureMQTTOperationTimeout(config.MQTT_OPER_TIMEOUT)
pycomAwsMQTTClient.configureLastWill(config.LAST_WILL_TOPIC, config.LAST_WILL_MSG, 1)

#Connect to MQTT Host
if pycomAwsMQTTClient.connect():
    print('AWS connection succeeded')

# Subscribe to topic
pycomAwsMQTTClient.subscribe(config.TOPIC, 1, customCallback)
time.sleep(2)

# Send message to host
loopCount = 0
while loopCount < 8:
    pycomAwsMQTTClient.publish(config.TOPIC, "New Message " + str(loopCount), 1)
    loopCount += 1
    time.sleep(5.0)
```

Shadow updater (`main.py`)

```python
# user specified callback functions
def customShadowCallback_Update(payload, responseStatus, token):
    if responseStatus == "timeout":
        print("Update request " + token + " time out!")
    if responseStatus == "accepted":
        payloadDict = json.loads(payload)
        print("Update request with token: " + token + " accepted!")
        print("property: " + str(payloadDict["state"]["desired"]["property"]))
    if responseStatus == "rejected":
        print("Update request " + token + " rejected!")

def customShadowCallback_Delete(payload, responseStatus, token):
    if responseStatus == "timeout":
        print("Delete request " + token + " time out!")
    if responseStatus == "accepted":
        print("Delete request with token: " + token + " accepted!")
    if responseStatus == "rejected":
        print("Delete request " + token + " rejected!")

# configure the MQTT client
pycomAwsMQTTShadowClient = AWSIoTMQTTShadowClient(config.CLIENT_ID)
pycomAwsMQTTShadowClient.configureEndpoint(config.AWS_HOST, config.AWS_PORT)
pycomAwsMQTTShadowClient.configureCredentials(config.AWS_ROOT_CA, config.AWS_PRIVATE_K
EY, config.AWS_CLIENT_CERT)

pycomAwsMQTTShadowClient.configureConnectDisconnectTimeout(config.CONN_DISCONN_TIMEOUT
)
pycomAwsMQTTShadowClient.configureMQTTOperationTimeout(config.MQTT_OPER_TIMEOUT)

# Connect to MQTT Host
if pycomAwsMQTTShadowClient.connect():
    print('AWS connection succeeded')

deviceShadowHandler = pycomAwsMQTTShadowClient.createShadowHandlerWithName(config.THIN
G_NAME, True)

# Delete shadow JSON doc
deviceShadowHandler.shadowDelete(customShadowCallback_Delete, 5)

# Update shadow in a loop
loopCount = 0
while True:
    JSONPayload = '{"state":{"desired":{"property":' + str(loopCount) + '}}}'
    deviceShadowHandler.shadowUpdate(JSONPayload, customShadowCallback_Update, 5)
    loopCount += 1
    time.sleep(5)
```

Delta Listener (`main.py`)

```
# Custom Shadow callback
def customShadowCallback_Delta(payload, responseStatus, token):
    payloadDict = json.loads(payload)
    print("property: " + str(payloadDict["state"]["property"]))
    print("version: " + str(payloadDict["version"]))

    # configure the MQTT client
pycomAwsMQTTShadowClient = AWSIoTMQTTShadowClient(config.CLIENT_ID)
pycomAwsMQTTShadowClient.configureEndpoint(config.AWS_HOST, config.AWS_PORT)
pycomAwsMQTTShadowClient.configureCredentials(config.AWS_ROOT_CA, config.AWS_PRIVATE_K
EY, config.AWS_CLIENT_CERT)

pycomAwsMQTTShadowClient.configureConnectDisconnectTimeout(config.CONN_DISCONN_TIMEOUT
)
pycomAwsMQTTShadowClient.configureMQTTOperationTimeout(config.MQTT_OPER_TIMEOUT)

# Connect to MQTT Host
if pycomAwsMQTTShadowClient.connect():
    print('AWS connection succeeded')

deviceShadowHandler = pycomAwsMQTTShadowClient.createShadowHandlerWithName(config.THIN
G_NAME, True)

# Listen on deltas
deviceShadowHandler.shadowRegisterDeltaCallback(customShadowCallback_Delta)

# Loop forever
while True:
    time.sleep(1)
```

ADC

This example is a simple ADC sample. For more information please see `ADC` .

```
from machine import ADC
adc = ADC(0)
adc_c = adc.channel(pin='P13')
adc_c()
adc_c.value()
```

Calibration

Currently the ESP32's ADC is not calibrated from the factory. This means it must be calibrated each time you wish to use it. To do this you must firstly measure the internal voltage reference. The following code will connect the 1.1v reference to `P22`

```
from machine import ADC
adc = ADC()

# Output Vref of P22
adc.vref_to_pin('P22')
```

Now that the voltage reference is externally accessible you should measure it with the most accurate voltmeter you have access to. Note down the reading in millivolts, e.g. `1120` . To disconnect the 1.1v reference from `P22` please reset your module. You can now calibrate the ADC by telling it the true value of the internal reference. You should then check your calibration by connecting the ADC to a known voltage source.

```
# Set calibration - see note above
adc.vref(1100)

# Check calibration by reading a known voltage
adc_c = adc.channel(pin='P16', attn=ADC.ATTN_11DB)
print(adc_c.voltage())
```

I2C

The following example receives data from a light sensor using I2C. Sensor used is the BH1750FVI Digital Light Sensor.

```
import time
from machine import I2C
import bh1750fvi

i2c = I2C(0, I2C.MASTER, baudrate=100000)
light_sensor = bh1750fvi.BH1750FVI(i2c, addr=i2c.scan()[0])

while(True):
        data = light_sensor.read()
        print(data)
    time.sleep(1)
```

Drivers for the BH1750FVI

Place this sample code into a file named `bh1750fvi.py`. This can then be imported as a library.

```
# Simple driver for the BH1750FVI digital light sensor

class BH1750FVI:
    MEASUREMENT_TIME = const(120)

    def __init__(self, i2c, addr=0x23, period=150):
        self.i2c = i2c
        self.period = period
        self.addr = addr
        self.time = 0
        self.value = 0
        self.i2c.writeto(addr, bytes([0x10])) # start continuos 1 Lux readings every 1
20ms

    def read(self):
        self.time += self.period
        if self.time >= MEASUREMENT_TIME:
            self.time = 0
            data = self.i2c.readfrom(self.addr, 2)
            self.value = (((data[0] << 8) + data[1]) * 1200) // 1000
        return self.value
```

Light sensor and LoRa

This is the same code, with added LoRa connectivity, sending the lux value from the light sensor to another LoRa enabled device.

```
import socket
import time
import pycom
import struct
from network import LoRa
from machine import I2C
import bh1750fvi

LORA_PKG_FORMAT = "!BH"
LORA_CONFIRM_FORMAT = "!BB"

DEVICE_ID = 1

pycom.heartbeat(False)

lora = LoRa(mode=LoRa.LORA, tx_iq=True, region=LoRa.EU868)
lora_sock = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
lora_sock.setblocking(False)

i2c = I2C(0, I2C.MASTER, baudrate=100000)
light_sensor = bh1750fvi.BH1750FVI(i2c, addr=i2c.scan()[0])

while(True):
    msg = struct.pack(LORA_PKG_FORMAT, DEVICE_ID, light_sensor.read())
    lora_sock.send(msg)

    pycom.rgbled(0x150000)

    wait = 5
    while (wait > 0):
        wait = wait - 0.1
        time.sleep(0.1)
        recv_data = lora_sock.recv(64)

        if (len (recv_data) >= 2):
            status, device_id = struct.unpack(LORA_CONFIRM_FORMAT, recv_data)

            if (device_id == DEVICE_ID and status == 200):
                pycom.rgbled(0x001500)
                wait = 0

    time.sleep(1)
```

Onewire Driver

This tutorial explains how to connect and read data from a DS18x20 temperature sensor. The onewire library is also available at the pycom-libraries GitHub Repository.

Basic usage

```
import time
from machine import Pin
from onewire import DS18X20
from onewire import OneWire

# DS18B20 data line connected to pin P10
ow = OneWire(Pin('P10'))
temp = DS18X20(ow)

while True:
    print(temp.read_temp_async())
    time.sleep(1)
    temp.start_conversion()
    time.sleep(1)
```

Library

```
#!/usr/bin/env python3

"""
OneWire library for MicroPython
"""

import time
import machine

class OneWire:
    CMD_SEARCHROM = const(0xf0)
    CMD_READROM = const(0x33)
    CMD_MATCHROM = const(0x55)
    CMD_SKIPROM = const(0xcc)

    def __init__(self, pin):
        self.pin = pin
        self.pin.init(pin.OPEN_DRAIN, pin.PULL_UP)

    def reset(self):
        """
```

```
        Perform the onewire reset function.
        Returns True if a device asserted a presence pulse, False otherwise.
        """
        sleep_us = time.sleep_us
        disable_irq = machine.disable_irq
        enable_irq = machine.enable_irq
        pin = self.pin

        pin(0)
        sleep_us(480)
        i = disable_irq()
        pin(1)
        sleep_us(60)
        status = not pin()
        enable_irq(i)
        sleep_us(420)
        return status

    def read_bit(self):
        sleep_us = time.sleep_us
        enable_irq = machine.enable_irq
        pin = self.pin

        pin(1) # half of the devices don't match CRC without this line
        i = machine.disable_irq()
        pin(0)
        sleep_us(1)
        pin(1)
        sleep_us(1)
        value = pin()
        enable_irq(i)
        sleep_us(40)
        return value

    def read_byte(self):
        value = 0
        for i in range(8):
            value |= self.read_bit() << i
        return value

    def read_bytes(self, count):
        buf = bytearray(count)
        for i in range(count):
            buf[i] = self.read_byte()
        return buf

    def write_bit(self, value):
        sleep_us = time.sleep_us
        pin = self.pin

        i = machine.disable_irq()
        pin(0)
        sleep_us(1)
```

```
        pin(value)
        sleep_us(60)
        pin(1)
        sleep_us(1)
        machine.enable_irq(i)

    def write_byte(self, value):
        for i in range(8):
            self.write_bit(value & 1)
            value >>= 1

    def write_bytes(self, buf):
        for b in buf:
            self.write_byte(b)

    def select_rom(self, rom):
        """
        Select a specific device to talk to. Pass in rom as a bytearray (8 bytes).
        """
        self.reset()
        self.write_byte(CMD_MATCHROM)
        self.write_bytes(rom)

    def crc8(self, data):
        """
        Compute CRC
        """
        crc = 0
        for i in range(len(data)):
            byte = data[i]
            for b in range(8):
                fb_bit = (crc ^ byte) & 0x01
                if fb_bit == 0x01:
                    crc = crc ^ 0x18
                crc = (crc >> 1) & 0x7f
                if fb_bit == 0x01:
                    crc = crc | 0x80
                byte = byte >> 1
        return crc

    def scan(self):
        """
        Return a list of ROMs for all attached devices.
        Each ROM is returned as a bytes object of 8 bytes.
        """
        devices = []
        diff = 65
        rom = False
        for i in range(0xff):
            rom, diff = self._search_rom(rom, diff)
            if rom:
                devices += [rom]
            if diff == 0:
```

```
                break
        return devices

    def _search_rom(self, l_rom, diff):
        if not self.reset():
            return None, 0
        self.write_byte(CMD_SEARCHROM)
        if not l_rom:
            l_rom = bytearray(8)
        rom = bytearray(8)
        next_diff = 0
        i = 64
        for byte in range(8):
            r_b = 0
            for bit in range(8):
                b = self.read_bit()
                if self.read_bit():
                    if b: # there are no devices or there is an error on the bus
                        return None, 0
                else:
                    if not b: # collision, two devices with different bit meaning
                        if diff > i or ((l_rom[byte] & (1 << bit)) and diff != i):
                            b = 1
                            next_diff = i
                self.write_bit(b)
                if b:
                    r_b |= 1 << bit
                i -= 1
            rom[byte] = r_b
        return rom, next_diff

class DS18X20(object):
    def __init__(self, onewire):
        self.ow = onewire
        self.roms = [rom for rom in self.ow.scan() if rom[0] == 0x10 or rom[0] == 0x28
]

    def isbusy(self):
        """
        Checks wether one of the DS18x20 devices on the bus is busy
        performing a temperature conversion
        """
        return not self.ow.read_bit()

    def start_conversion(self, rom=None):
        """
        Start the temp conversion on one DS18x20 device.
        Pass the 8-byte bytes object with the ROM of the specific device you want to r
ead.
        If only one DS18x20 device is attached to the bus you may omit the rom paramet
er.
        """
        rom = rom or self.roms[0]
```

```
        ow = self.ow
        ow.reset()
        ow.select_rom(rom)
        ow.write_byte(0x44)  # Convert Temp

    def read_temp_async(self, rom=None):
        """
        Read the temperature of one DS18x20 device if the conversion is complete,
        otherwise return None.
        """
        if self.isbusy():
            return None
        rom = rom or self.roms[0]
        ow = self.ow
        ow.reset()
        ow.select_rom(rom)
        ow.write_byte(0xbe)  # Read scratch
        data = ow.read_bytes(9)
        return self.convert_temp(rom[0], data)

    def convert_temp(self, rom0, data):
        """
        Convert the raw temperature data into degrees celsius and return as a fixed po
int with 2 decimal places.
        """
        temp_lsb = data[0]
        temp_msb = data[1]
        if rom0 == 0x10:
            if temp_msb != 0:
                # convert negative number
                temp_read = temp_lsb >> 1 | 0x80  # truncate bit 0 by shifting, fill h
igh bit with 1.
                temp_read = -((~temp_read + 1) & 0xff) # now convert from two's comple
ment
            else:
                temp_read = temp_lsb >> 1  # truncate bit 0 by shifting
            count_remain = data[6]
            count_per_c = data[7]
            temp = 100 * temp_read - 25 + (count_per_c - count_remain) // count_per_c
            return temp
        elif rom0 == 0x28:
            return (temp_msb << 8 | temp_lsb) * 100 // 16
        else:
            assert False
```

Threading

MicroPython supports spawning threads by the `_thread` module. The following example demonstrates the use of this module. A thread is simply defined as a function that can receive any number of parameters. Below 3 threads are started, each one perform a print at a different interval.

```
import _thread
import time

def th_func(delay, id):
    while True:
        time.sleep(delay)
        print('Running thread %d' % id)

for i in range(3):
    _thread.start_new_thread(th_func, (i + 1, i))
```

Using Locks:

```
import _thread

a_lock = _thread.allocate_lock()

with a_lock:
    print("a_lock is locked while this executes")
```

RGB LED

By default the heartbeat LED flashes in blue colour once every 4s to signal that the system is alive. This can be overridden through the `pycom` module.

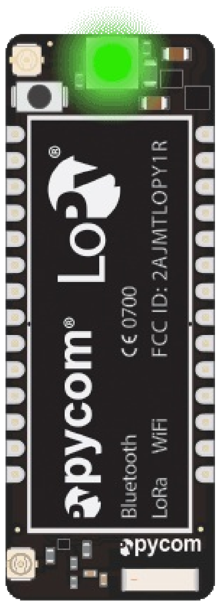
```
import pycom

pycom.heartbeat(False)
pycom.rgbled(0xff00)           # turn on the RGB LED in green colour
```

The heartbeat LED is also used to indicate that an error was detected.

The following piece of code uses the RGB LED to make a traffic light that runs for 10 cycles.

```
import pycom
import time

pycom.heartbeat(False)
for cycles in range(10): # stop after 10 cycles
    pycom.rgbled(0x007f00) # green
    time.sleep(5)
    pycom.rgbled(0x7f7f00) # yellow
    time.sleep(1.5)
    pycom.rgbled(0x7f0000) # red
    time.sleep(4)
```

Here is the expected result:



Timers

Detailed information about this class can be found in `Timer` .

Chronometer

The Chronometer can be used to measure how much time has elapsed in a block of code. The following example uses a simple stopwatch.

```
from machine import Timer
import time

chrono = Timer.Chrono()

chrono.start()
time.sleep(1.25) # simulate the first lap took 1.25 seconds
lap = chrono.read() # read elapsed time without stopping
time.sleep(1.5)
chrono.stop()
total = chrono.read()

print()
print("\nthe racer took %f seconds to finish the race" % total)
print("  %f seconds in the first lap" % lap)
print("  %f seconds in the last lap" % (total - lap))
```

Alarm

The Alarm can be used to get interrupts at a specific interval. The following code executes a callback every second for 10 seconds.

```python
from machine import Timer

class Clock:

    def __init__(self):
        self.seconds = 0
        self.__alarm = Timer.Alarm(self._seconds_handler, 1, periodic=True)

    def _seconds_handler(self, alarm):
        self.seconds += 1
        print("%02d seconds have passed" % self.seconds)
        if self.seconds == 10:
            alarm.callback(None) # stop counting after 10 seconds

clock = Clock()
```

There are no restrictions to what can be done in an interrupt. For example, it is possible to even do network requests with an interrupt. However, it is important to keep in mind that interrupts are handled sequentially, so it's good practice to keep them short. More information can be found in `Interrupt Handling`.

PIR Sensor

This code reads PIR sensor triggers from this simple PIR sensor and sends an HTTP request for every trigger, in this case to a Domoticz installation. When motion is constantly detected, this PIR sensor keeps the pin high, in which case this code will keep sending HTTP requests every 10 seconds (configurable with the hold_time variable).

Main (`main.py`)

```
import time
from network import WLAN
from machine import Pin
from domoticz import Domoticz

wl = WLAN(WLAN.STA)
d = Domoticz("<ip>", 8080 ,"<hash>")

#config
hold_time_sec = 10

#flags
last_trigger = -10

pir = Pin('G4',mode=Pin.IN, pull=Pin.PULL_UP)

# main loop
print("Starting main loop")
while True:
    if pir() == 1:
        if time.time() - last_trigger > hold_time_sec:
            last_trigger = time.time()
            print("Presence detected, sending HTTP request")
            try:
                return_code = d.setVariable('Presence:LivingRoom','1')
                print("Request result: "+str(return_code))
            except Exception as e:
                print("Request failed")
                print(e)
    else:
        last_trigger = 0
        print("No presence")

    time.sleep_ms(500)

print("Exited main loop")
```

Boot (`boot.py`)

For more WiFi scripts, see the wlan step by step tutorial.

```
import os
import machine

uart = machine.UART(0, 115200)
os.dupterm(uart)

known_nets = {
    'NetworkID':        {'pwd': '<password>', 'wlan_config':  ('10.0.0.8', '255.255.0.
0', '10.0.0.1', '10.0.0.1')},
}

from network import WLAN
wl = WLAN()


if machine.reset_cause() != machine.SOFT_RESET:

    wl.mode(WLAN.STA)
    original_ssid = wl.ssid()
    original_auth = wl.auth()

    print("Scanning for known wifi nets")
    available_nets = wl.scan()
    nets = frozenset([e.ssid for e in available_nets])

    known_nets_names = frozenset([key for key in known_nets])
    net_to_use = list(nets & known_nets_names)
    try:
        net_to_use = net_to_use[0]
        net_properties = known_nets[net_to_use]
        pwd = net_properties['pwd']
        sec = [e.sec for e in available_nets if e.ssid == net_to_use][0]
        if 'wlan_config' in net_properties:
            wl.ifconfig(config=net_properties['wlan_config'])
        wl.connect(net_to_use, (sec, pwd), timeout=10000)
        while not wl.isconnected():
            machine.idle() # save power while waiting
        print("Connected to "+net_to_use+" with IP address:" + wl.ifconfig()[0])

    except Exception as e:
        print("Failed to connect to any known network, going into AP mode")
        wl.init(mode=WLAN.AP, ssid=original_ssid, auth=original_auth, channel=6, anten
na=WLAN.INT_ANT)
```

Domoticz Wrapper (`domoticz.py`)

```
import socket
class Domoticz:

    def __init__(self, ip, port,  basic):
        self.basic = basic
        self.ip = ip
        self.port = port

    def setLight(self, idx, command):
        return self.sendRequest("type=command&param=switchlight&idx="+idx+"&switchcmd="
+command)

    def setVariable(self, name, value):
        return self.sendRequest("type=command&param=updateuservariable&vtype=0&vname="
+name+"&vvalue="+value)

    def sendRequest(self, path):
        try:
            s = socket.socket()
            s.connect((self.ip,self.port))
            s.send(b"GET /json.htm?"+path+" HTTP/1.1\r\nHost: pycom.io\r\nAuthorizatio
n: Basic "+self.basic+"\r\n\r\n")
            status = str(s.readline(), 'utf8')
            code = status.split(" ")[1]
            s.close()
            return code

        except Exception:
            print("HTTP request failed")
            return 0
```

Modbus Protocol

Modbus is a messaging protocol that defines the packet structure for transferring data between devices in a master/slave architecture. The protocol is independent of the transmission medium and is usually transmitted over TCP (MODBUS TCP) or serial communication (MODBUS RTU). Modbus is intended as a request/reply protocol and delivers services specified by function codes. The function code in the request tells the addressed slave what kind of action to perform. The function codes most commonly supported by devices are listed below.

Function Name	Function Code
Read Coils	0x01
Read Discrete Inputs	0x02
Read Holding Registers	0x03
Read Input Registers	0x04
Write Single Coil	0x05
Write Single Register	0x06
Write Multiple Coils	0x0F
Write Multiple Registers	0x10

For more information on the MODBUS RTU see the following PDF File. Information on the MODBUS TCP can be found here.

Pycom Modbus Library

Python libraries and sample code that support Modbus TCP and Modbus RTU are available at the following GitHub Repository. To use this library, connect to the target Pycom device via ftp and upload the uModbus folder to `/flash`. A description of the supported function codes is found below.

Read Coils

This function code requests the status (ON/OFF) of discrete coils on a remote device. The slave device address, the address of the first coil and the number of coils must be specified in the request. The address of the first coil is 0 and a maximum of 2000 contiguous coils can be read. Python sample code is shown below.

```
slave_addr=0x0A
starting_address=0x00
coil_quantity=100

coil_status = modbus_obj.read_coils(slave_addr, starting_address, coil_quantity)
print('Coil status: ' + ' '.join('{:d}'.format(x) for x in coil_status))
```

Read Discrete Inputs

This command is used to read the status (ON/OFF) of discrete inputs on a remote device. The slave address, the address of the first input, and the quantity of inputs to be read must be specified. The address of the first input is 0 and a maximum of 2000 continuous inputs can be read. The Python sample code is shown below.

```
slave_addr=0x0A
starting_address=0x0
input_quantity=100

input_status = modbus_obj.read_discrete_inputs(slave_addr, starting_address, input_qua
ntity)
print('Input status: ' + ' '.join('{:d}'.format(x) for x in input_status))
```

Read Holding Registers

This function code is used to read the contents of analogue output holding registers. The slave address, the starting register address, the number of registers to read and the sign of the data must be specified. Register addresses start at 0 and a maximum of 125 continuous registers can be read.

```
slave_addr=0x0A
starting_address=0x00
register_quantity=100
signed=True

register_value = modbus_obj.read_holding_registers(slave_addr, starting_address, regis
ter_quantity, signed)
print('Holding register value: ' + ' '.join('{:d}'.format(x) for x in register_value))
```

Read Input Registers

This command is used to read up to 125 continuous input registers on a remote device. The slave address, the starting register address, the number of input registers and the sign of the data must be specified. The address of the first input registers is 0.

```
slave_addr=0x0A
starting_address=0x00
register_quantity=100
signed=True

register_value = modbus_obj.read_input_registers(slave_addr, starting_address, registe
r_quantity, signed)
print('Input register value: ' + ' '.join('{:d}'.format(x) for x in register_value))
```

Write Single Coil

This function code is used to write the state of a discrete coil on a remote device. A value of `0xFF00` means the coil should be set to ON, while a value of `0x0000` means the coil should be set to OFF. The Python sample code to set the coil at address `0x00`, to an ON state is shown below.

```
slave_addr=0x0A
output_address=0x00
output_value=0xFF00

return_flag = modbus_obj.write_single_coil(slave_addr, output_address, output_value)
output_flag = 'Success' if return_flag else 'Failure'
print('Writing single coil status: ' + output_flag)
```

Write Single Register

This command is used to write the contents of an analog output holding register on a remote device. The slave address, the register address, the register value, and the signature of the data must be specified. As for all the other commands, the register addresses start from 0.

```
slave_addr=0x0A
register_address=0x01
register_value=-32768
signed=True

return_flag = modbus_obj.write_single_register(slave_addr, register_address, register_
value, signed)
output_flag = 'Success' if return_flag else 'Failure'
print('Writing single coil status: ' + output_flag)
```

Write Multiple Coils

This function code is used to set a continuous sequence of coils, in a remote device, to either ON or OFF. The slave address, the starting address of the coils and an array with the coil states must be specified.

```
slave_addr=0x0A
starting_address=0x00
output_values=[1,1,1,0,0,1,1,1,0,0,1,1,1]

return_flag = modbus_obj.write_multiple_coils(slave_addr, starting_address, output_val
ues)
output_flag = 'Success' if return_flag else 'Failure'
print('Writing multiple coil status: ' + output_flag)
```

Write Multiple Registers

This command is used to write the contents of a continuous sequence of analogue registers on a remote device. The slave address, the starting register address, the register values, and the signature of the data must be specified. The address of the first register is 0 and a maximum of 125 register values can be written. The Python sample code is shown below.

```
slave_addr=0x0A
register_address=0x01
register_values=[2, -4, 6, -256, 1024]
signed=True

return_flag = modbus_obj.write_multiple_registers(slave_addr, register_address, regist
er_values, signed)
output_flag = 'Success' if return_flag else 'Failure'
print('Writing multiple register status: ' + output_flag)
```

Overview

Pycom modules come with the ability to update the devices firmware, while it is still running, we call this an "over the air" (OTA) update. The pycom library provides several functions to achieve this. This example will demonstrate how you could potentially use this functionality to update deployed devices. The full source code of this example can be found here.

Method

Here we will describe one possible update methodology you could use that is implemented by this example.

Imagine you a smart metering company and you wish to roll out an update for your Pycom based smart meter. These meters usually send data back via LoRa. Unfortunately LoRa downlink messages have a very limited size and several hundred if not thousand would be required to upload a complete firmware image. To get around this you can have your devices sending their regular data via LoRa and when they receive a special command via a downlink message, the devices will connect to a WiFi network. It is unfeasible to ask customers to allow your device to connect to their home network so instead this network could be provided by a vehicle. This vehicle will travel around a certain geographic area in which the devices have been sent the special downlink message to initiate the update. The devices will look for the WiFi network being broadcast by the vehicle and connect. The devices will then connect to a server running on this WiFi network. This server (also shown in this example) will generate manifest files that instruct the device on what it should update, and where to get the update data from.

Server

Code available here.

This script runs a HTTP server on port `8000` that provisions over the air (OTA) update manifests in JSON format as well as serving the update content. This script should be run in a directory that contains every version of the end devices code, in the following structure:

```
- server directory
  |- this_script.py
  |- 1.0.0
  |  |- flash
  |  |   |- lib
  |  |   |  |- lib_a.py
  |  |   |- main.py
  |  |   |- boot.py
  |  |- sd
  |     |- some_asset.txt
  |     |- asset_that_will_be_removed.wav
  |- 1.0.1
  |  |- flash
  |  |   |- lib
  |  |   |  |- lib_a.py
  |  |   |  |- new_lib.py
  |  |   |- main.py
  |  |   |- boot.py
  |  |- sd
  |     |- some_asset.txt
  |- firmware_1.0.0.bin
  |- firmware_1.0.1.bin
```

The top level directory that contains this script can contain one of two things:

- Update directory: These should be named with a version number compatible with the python LooseVersion versioning scheme (http://epydoc.sourceforge.net/stdlib/distutils.version.LooseVersion-class.html). They should contain the entire file system of the end device for the corresponding version number.
- Firmware: These files should be named in the format `firmare_VERSION.bin`, where VERSION is a a version number compatible with the python LooseVersion versioning scheme (http://epydoc.sourceforge.net/stdlib/distutils.version.LooseVersion-class.html). This file should be in the format of the `appimg.bin` created by the Pycom firmware build scripts.

How to use

Once the directory has been setup as described above you simply need to start this script using python3. Once started this script will run a HTTP server on port `8000` (this can be changed by changing the PORT variable). This server will serve all the files in directory as expected along with one additional special file, `manifest.json`. This file does not exist on the file system but is instead generated when requested and contains the required changes to bring the end device from its current version to the latest available version. You can see an example of this by pointing your web browser at:

`http://127.0.0.1:8000/manifest.json?current_ver=1.0.0`

The `current_ver` field at the end of the URL should be set to the current firmware version of the end device. The generated manifest will contain lists of which files are new, have changed or need to be deleted along with SHA1 hashes of the files. Below is an example of what such a manifest might look like:

```
{
   "delete": [
      "flash/old_file.py",
      "flash/other_old_file.py"
   ],
   "firmware": {
       "URL": "http://192.168.1.144:8000/firmware_1.0.1b.bin",
       "hash": "ccc6914a457eb4af8855ec02f6909316526bdd08"
   },
   "new": [
       {
           "URL": "http://192.168.1.144:8000/1.0.1b/flash/lib/new_lib.py",
           "dst_path": "flash/lib/new_lib.py",
           "hash": "1095df8213aac2983efd68dba9420c8efc9c7c4a"
       }
   ],
   "update": [
       {
           "URL": "http://192.168.1.144:8000/1.0.1b/flash/changed_file.py",
           "dst_path": "flash/changed_file.py",
           "hash": "1095df8213aac2983efd68dba9420c8efc9c7c4a"
       }
   ],
   "version": "1.0.1b"
}
```

The manifest contains the following fields:

- `delete` : A list of paths to files which are no longer needed
- `firmware` : The URL and SHA1 hash of the firmware image
- `new` : the URL, path on end device and SHA1 hash of all new files
- `update` : the URL, path on end device and SHA1 hash of all files which existed before but have changed.
- `version` : The version number that this manifest will update the client to
- `previous_version` : The version the client is currently on before applying this update

Note: The version number of the files might not be the same as the firmware. The highest available version number, higher than the current client version is used for both firmware and files. This may differ between the two.

In order for the URL's to be properly formatted you are required to send a "host" header along with your HTTP get request e.g:

```
GET /manifest.json?current_ver=1.0.0 HTTP/1.0\r\nHost: 192.168.1.144:8000\r\n\r\n
```

Client Library

A MicroPyton library for interfacing with the server described above is available here.

This library is split into two layers. The top level `OTA` class implements all the high level functionality such as parsing the JSON file, making back copies of files being updated incase the update fails, etc. The layer of the library is agnostic to your chosen transport method. Below this is the `WiFiOTA` class. This class implements the actual transport mechanism of how the device fetches the files and update manifest (via WiFi as the class name suggests). The reason for this split is so that the high level functionality can be reused regardless of what transport mechanism you end up using. This could be implemented on top of Bluetooth for example, or the sever changed from HTTP to FTP.

Although the above code is functional, it is provided only as an example of how an end user might implement a OTA update mechanism. It is not 100% feature complete e.g. even though it does backup previous versions of files, the roll back procedure is not implemented. This is left of the end user to do.

Example

Below is am example implementing the methodology previously explained in this tutorial to initiate an OTA update.

The example below will only work on a Pycom device with LoRa capabilities. If want to test it out on a device without LoRa functionality then simply comment out any code relating to LoRa. Leaving just the `WiFiOTA` initialisation and they `ota.connect()` and `ota.update()`

```
from network import LoRa, WLAN
import socket
import time
from OTA import WiFiOTA
from time import sleep
import pycom
import ubinascii

from config import WIFI_SSID, WIFI_PW, SERVER_IP

# Turn on GREEN LED
pycom.heartbeat(False)
pycom.rgbled(0xff00)

# Setup OTA
ota = WiFiOTA(WIFI_SSID,
              WIFI_PW,
              SERVER_IP,  # Update server address
              8000)  # Update server port

# Turn off WiFi to save power
w = WLAN()
w.deinit()

# Initialise LoRa in LORAWAN mode.
lora = LoRa(mode=LoRa.LORAWAN, region=LoRa.EU868)

app_eui = ubinascii.unhexlify('70B3D57ED0008CD6')
app_key = ubinascii.unhexlify('B57F36D88691CEC5EE8659320169A61C')

# join a network using OTAA (Over the Air Activation)
lora.join(activation=LoRa.OTAA, auth=(app_eui, app_key), timeout=0)

# wait until the module has joined the network
while not lora.has_joined():
    time.sleep(2.5)
    print('Not yet joined...')

# create a LoRa socket
s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)

# set the LoRaWAN data rate
s.setsockopt(socket.SOL_LORA, socket.SO_DR, 5)

# make the socket blocking
# (waits for the data to be sent and for the 2 receive windows to expire)
s.setblocking(True)

while True:
    # send some data
    s.send(bytes([0x04, 0x05, 0x06]))
```

```
    # make the socket non-blocking
    # (because if there's no data received it will block forever...)
    s.setblocking(False)

    # get any data received (if any...)
    data = s.recv(64)

    # Some sort of OTA trigger
    if data == bytes([0x01, 0x02, 0x03]):
        print("Performing OTA")
        # Perform OTA
        ota.connect()
        ota.update()

    sleep(5)
```

RMT

Detailed information about this class can be found in `RMT`.

The RMT (Remote Control) peripheral of the ESP32 is primarily designed to send and receive infrared remote control signals that use on-off-keying of a carrier frequency, but due to its design it can be used to generate various types of signals, this class will allow you to do this.

The RMT has 7 channels, of which 5 are available and can be mapped to any GPIO pin (*Note:* Pins `P13` - `P18` can only be used as inputs).

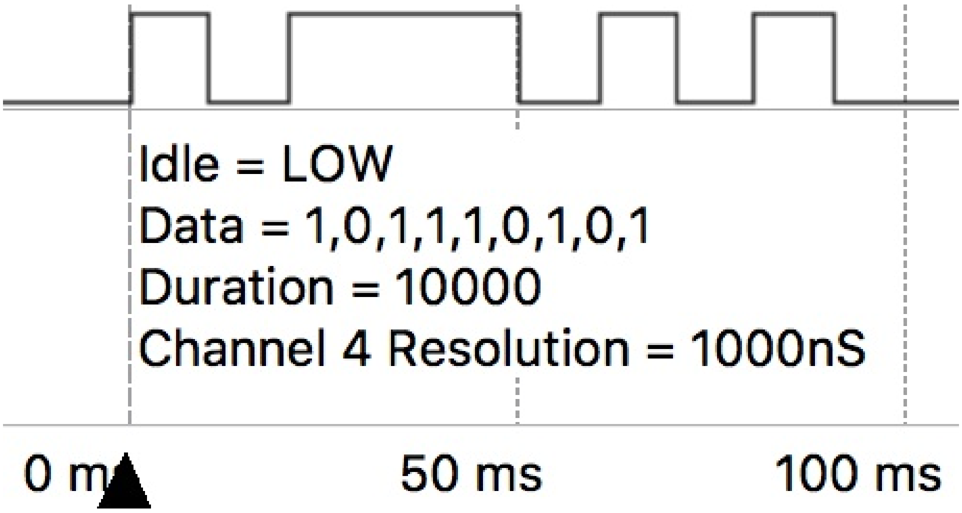
Channel	Resolution	Maximum Pulse Width
0	Used by on-board LED	
1	Used by `pycom.pulses_get()`	
2	100nS	3.2768 ms
3	100nS	3.2768 ms
4	1000nS	32.768 ms
5	1000nS	32.768 ms
6	3125nS	102.4 ms
7	3125nS	102.4 ms

Transmitting

The following examples create an RMT object on channel 4, configure it for transmission and send some data in various forms. The resolution of channel 4 is 1000 nano seconds, the given values are interpreted accordingly.

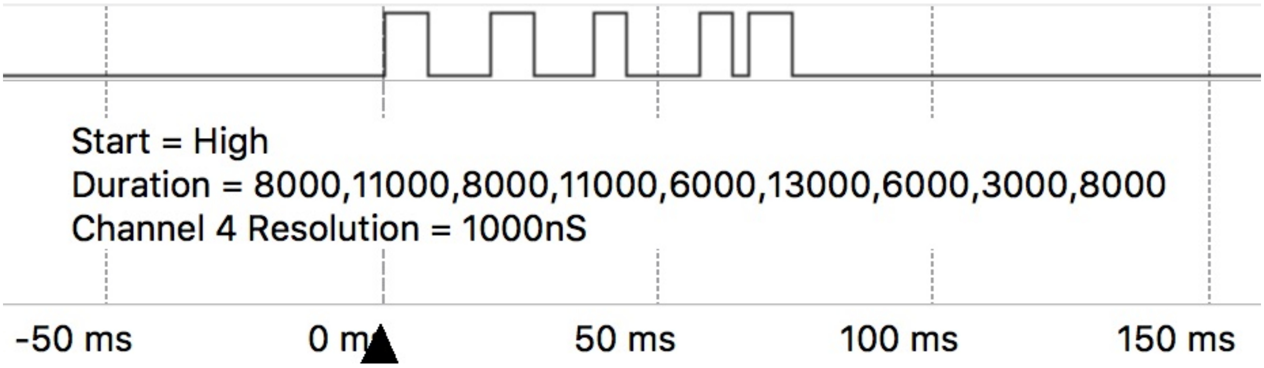
In this first example, we define the signal as a tuple of binary values that define the shape of the desired signal along with the duration of a bit.

```
from machine import RMT
# Map RMT channel 4 to P21, when the RMT is idle, it will output LOW
rmt = RMT(channel=4, gpio="P21", tx_idle_level=RMT.LOW)

# Produces the pattern shown in data, where each bit lasts
# duration * channel resolution = 10000 * 1000ns = 10ms
data = (1,0,1,1,1,0,1,0,1)
duration = 10000
rmt.pulses_send(duration, data)
```



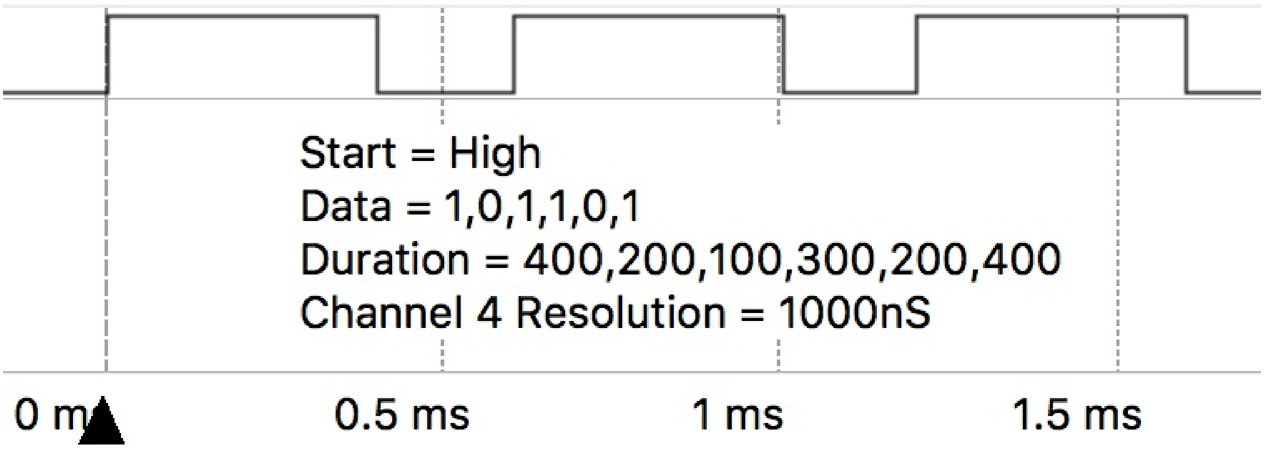
In this example we define the signal by a tuple of durations and what state the signal starts in.

```
from machine import RMT
# Map RMT channel 4 to P21, when the RMT is idle, it will output LOW
rmt = RMT(channel=4, gpio="P21", tx_idle_level=RMT.LOW)

# The list of durations for each pulse to be, these are in units of the channels
# resolution:
#    duration = Desired pulse length / Channel Resolution
duration = (8000,11000,8000,11000,6000,13000,6000,3000,8000)

# `start_level` defines if the signal starts off as LOW or HIGH, it will then
# toggle state between each duration
rmt.pulses_send(duration, start_level=RMT.HIGH)
```



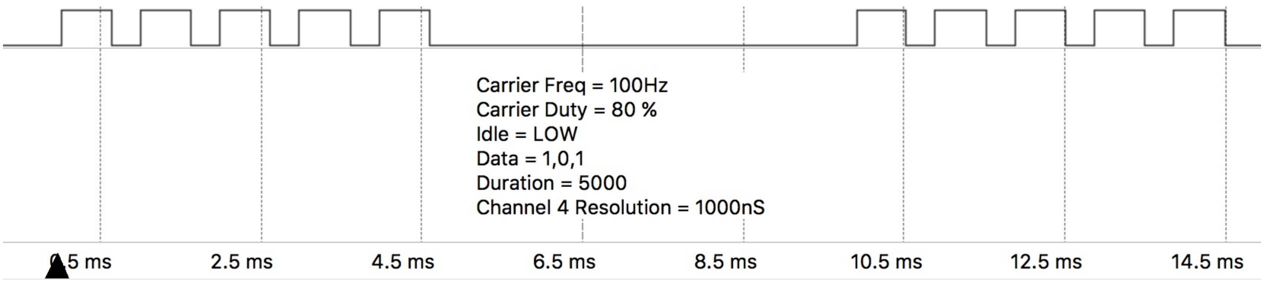
This third example, is a combination of the above two styles of defining a signal. Each pulse has a defined duration as well as a state. This is useful if you don't always want the signal to toggle state.

```
from machine import RMT
# Map RMT channel 4 to P21, when the RMT is idle, it will output LOW
rmt = RMT(channel=4, gpio="P21", tx_idle_level=RMT.LOW)

# Produces the pattern shown in data, where each bit lasts
# duration[i] * channel resolution = duration[i] * 1000ns
data = (1,0,1,1,0,1)
duration = (400,200,100,300,200,400)
rmt.pulses_send(duration, data)
```



The following example creates an RMT object on channel 4 and configures it for transmission with carrier modulation.

```
from machine import RMT
rmt = RMT(channel=4,
          gpio="P21",
          tx_idle_level=RMT.LOW,
          # Carrier = 100Hz, 80% duty, modules HIGH signals
          tx_carrier = (100, 70, RMT.HIGH))
data = (1,0,1)
duration = 10000
rmt.pulses_send(duration, data)
```



The following example creates an RMT object on channel 2, configures it for receiving, then waits for the first, undefined number of pulses without timeout

```
from machine import RMT
rmt = machine.RMT(channel=2)
rmt.init(gpio="P21", rx_idle_threshold=1000)

data = rmt.pulses_get()
```

If `tx_idle_level` is not set to the opposite of the third value in the `tx_carrier` tuple, the carrier wave will continue to be generated when the RMT channel is idle.

Receiving

The following example creates an RMT object on channel 2, configures it for receiving a undefined number of pulses, then waits maximum of 1000us for the first pulse.

```
from machine import RMT
# Sets RMT channel 2 to P21 and sets the maximum length of a valid pulse to
# 1000*channel resolution = 1000 * 100ns = 100us
rmt = machine.RMT(channel=2, gpio="P21", rx_idle_threshold=1000)
rmt.init()

# Get a undefined number of pulses, waiting a maximum of 500us for the first
# pulse (unlike other places where the absolute duration was based on the RMT
# channels resolution, this value is in us) until a pulse longer than
# rx_idle_threshold occurs.
data = rmt.pulses_get(timeout=500)
```

The following example creates an RMT object on channel 2, configures it for receiving, filters out pulses with width < 20*100 nano seconds, then waits for 100 pulses

```python
from machine import RMT

rmt = machine.RMT(channel=2,  # Resolution = 100ns
                  gpio="P21",
                  # Longest valid pulse = 1000*100ns = 100us
                  rx_idle_threshold=1000,
                  # Filter out pulses shorter than 20*100ns = 2us
                  rx_filter_threshold=20)

# Receive 100 pulses, pulses shorter than 2us or longer than 100us will be
# ignored. That means if it receives 80 valid pulses but then the signal
# doesn't change for 10 hours and then 20 more pulses occur, this function
# will wait for 10h
data = rmt.pulses_get(pulses=100)
```

LoPy Tutorials

The following tutorials demonstrate the use of the LoRa functionality on the LoPy. LoRa can work in 2 different modes; **LoRa-MAC** (which we also call Raw-LoRa) and **LoRaWAN** mode.

LoRa-MAC mode basically accesses de radio directly and packets are sent using the LoRa modulation on the selected frequency without any headers, addressing information or encryption. Only a CRC is added at the tail of the packet and this is removed before the received frame is passed on to the application. This mode can be used to build any higher level protocol that can benefit from the long range features of the LoRa modulation. Typical uses cases include LoPy to LoPy direct communication and a LoRa packet forwarder.

LoRaWAN mode implements the full LoRaWAN stack for a class A device. It supports both OTAA and ABP connection methods, as well as advanced features like adding and removing custom channels to support "special" frequencies plans like the those used in New Zealand.

LoRa-MAC (Raw LoRa)

Basic LoRa connection example, sending and receiving data. In LoRa-MAC mode the LoRaWAN layer is bypassed and the radio is used directly. The data sent is not formatted or encrypted in any way, and no addressing information is added to the frame.

For the example below, you will need two LoPys. A `while` loop with a random delay time is used to minimise the chances of the 2 LoPy's transmitting at the same time. Run the code below on the 2 LoPy modules and you will see the word 'Hello' being received on both sides.

```
from network import LoRa
import socket
import machine
import time

# initialise LoRa in LORA mode
# Please pick the region that matches where you are using the device:
# Asia = LoRa.AS923
# Australia = LoRa.AU915
# Europe = LoRa.EU868
# United States = LoRa.US915
# more params can also be given, like frequency, tx power and spreading factor
lora = LoRa(mode=LoRa.LORA, region=LoRa.EU868)

# create a raw LoRa socket
s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)

while True:
    # send some data
    s.setblocking(True)
    s.send('Hello')

    # get any data received...
    s.setblocking(False)
    data = s.recv(64)
    print(data)

    # wait a random amount of time
    time.sleep(machine.rng() & 0x0F)
```

LoRaWAN (OTAA)

OTAA stands for Over The Air Authentication. With this method the LoPy sends a Join request to the LoRaWAN Gateway using the `APPEUI` and `APPKEY` provided. If the keys are correct the Gateway will reply to the LoPy with a join accept message and from that point on the LoPy is able to send and receive packets to/from the Gateway. If the keys are incorrect no response will be received and the `has_joined()` method will always return `False`.

The example below attempts to get any data received after sending the frame. Keep in mind that the Gateway might not be sending any data back, therefore we make the socket non-blocking before attempting to receive, in order to prevent getting stuck waiting for a packet that will never arrive.

```
from network import LoRa
import socket
import time
import ubinascii

# Initialise LoRa in LORAWAN mode.
# Please pick the region that matches where you are using the device:
# Asia = LoRa.AS923
# Australia = LoRa.AU915
# Europe = LoRa.EU868
# United States = LoRa.US915
lora = LoRa(mode=LoRa.LORAWAN, region=LoRa.EU868)

# create an OTAA authentication parameters
app_eui = ubinascii.unhexlify('ADA4DAE3AC12676B')
app_key = ubinascii.unhexlify('11B0282A189B75B0B4D2D8C7FA38548B')

# join a network using OTAA (Over the Air Activation)
lora.join(activation=LoRa.OTAA, auth=(app_eui, app_key), timeout=0)

# wait until the module has joined the network
while not lora.has_joined():
    time.sleep(2.5)
    print('Not yet joined...')

# create a LoRa socket
s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)

# set the LoRaWAN data rate
s.setsockopt(socket.SOL_LORA, socket.SO_DR, 5)

# make the socket blocking
# (waits for the data to be sent and for the 2 receive windows to expire)
s.setblocking(True)

# send some data
s.send(bytes([0x01, 0x02, 0x03]))

# make the socket non-blocking
# (because if there's no data received it will block forever...)
s.setblocking(False)

# get any data received (if any...)
data = s.recv(64)
print(data)
```

LoRaWAN (ABP)

ABP stands for Authentication By Personalisation. It means that the encryption keys are configured manually on the device and can start sending frames to the Gateway without needing a 'handshake' procedure to exchange the keys (such as the one performed during an OTAA join procedure).

The example below attempts to get any data received after sending the frame. Keep in mind that the Gateway might not be sending any data back, therefore we make the socket non-blocking before attempting to receive, in order to prevent getting stuck waiting for a packet that will never arrive.

```
from network import LoRa
import socket
import ubinascii
import struct

# Initialise LoRa in LORAWAN mode.
# Please pick the region that matches where you are using the device:
# Asia = LoRa.AS923
# Australia = LoRa.AU915
# Europe = LoRa.EU868
# United States = LoRa.US915
lora = LoRa(mode=LoRa.LORAWAN, region=LoRa.EU868)

# create an ABP authentication params
dev_addr = struct.unpack(">l", binascii.unhexlify('00000005'))[0]
nwk_swkey = ubinascii.unhexlify('2B7E151628AED2A6ABF7158809CF4F3C')
app_swkey = ubinascii.unhexlify('2B7E151628AED2A6ABF7158809CF4F3C')

# join a network using ABP (Activation By Personalization)
lora.join(activation=LoRa.ABP, auth=(dev_addr, nwk_swkey, app_swkey))

# create a LoRa socket
s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)

# set the LoRaWAN data rate
s.setsockopt(socket.SOL_LORA, socket.SO_DR, 5)

# make the socket blocking
# (waits for the data to be sent and for the 2 receive windows to expire)
s.setblocking(True)

# send some data
s.send(bytes([0x01, 0x02, 0x03]))

# make the socket non-blocking
# (because if there's no data received it will block forever...)
s.setblocking(False)

# get any data received (if any...)
data = s.recv(64)
print(data)
```

LoRa Nano-Gateway (Raw LoRa)

This example allows a raw LoRa connection between two LoPys (nodes) to a single LoPy acting as a Nano-Gateway.

For more information and discussions about this code, see this forum post.

Gateway Code

```
import socket
import struct
from network import LoRa

# A basic package header, B: 1 byte for the deviceId, B: 1 byte for the pkg size, %ds:
 Formatted string for string
_LORA_PKG_FORMAT = "!BB%ds"
# A basic ack package, B: 1 byte for the deviceId, B: 1 byte for the pkg size, B: 1 by
te for the Ok (200) or error messages
_LORA_PKG_ACK_FORMAT = "BBB"

# Open a LoRa Socket, use rx_iq to avoid listening to our own messages
# Please pick the region that matches where you are using the device:
# Asia = LoRa.AS923
# Australia = LoRa.AU915
# Europe = LoRa.EU868
# United States = LoRa.US915
lora = LoRa(mode=LoRa.LORA, rx_iq=True, region=LoRa.EU868)
lora_sock = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
lora_sock.setblocking(False)

while (True):
    recv_pkg = lora_sock.recv(512)
    if (len(recv_pkg) > 2):
        recv_pkg_len = recv_pkg[1]

        device_id, pkg_len, msg = struct.unpack(_LORA_PKG_FORMAT % recv_pkg_len, recv_
pkg)

# If the uart = machine.UART(0, 115200) and os.dupterm(uart) are set in the boot.py th
is print should appear in the serial port
        print('Device: %d - Pkg:  %s' % (device_id, msg))

        ack_pkg = struct.pack(_LORA_PKG_ACK_FORMAT, device_id, 1, 200)
        lora_sock.send(ack_pkg)
```

The `_LORA_PKG_FORMAT` is used as a method of identifying the different devices within a network. The `_LORA_PKG_ACK_FORMAT` is a simple `ack` package as a response to the nodes package.

Node

```
import os
import socket
import time
import struct
from network import LoRa

# A basic package header, B: 1 byte for the deviceId, B: 1 byte for the pkg size
_LORA_PKG_FORMAT = "BB%ds"
_LORA_PKG_ACK_FORMAT = "BBB"
DEVICE_ID = 0x01


# Open a Lora Socket, use tx_iq to avoid listening to our own messages
# Please pick the region that matches where you are using the device:
# Asia = LoRa.AS923
# Australia = LoRa.AU915
# Europe = LoRa.EU868
# United States = LoRa.US915
lora = LoRa(mode=LoRa.LORA, tx_iq=True, region=LoRa.EU868)
lora_sock = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
lora_sock.setblocking(False)

while(True):
    # Package send containing a simple string
    msg = "Device 1 Here"
    pkg = struct.pack(_LORA_PKG_FORMAT % len(msg), DEVICE_ID, len(msg), msg)
    lora_sock.send(pkg)

    # Wait for the response from the gateway. NOTE: For this demo the device does an i
nfinite loop for while waiting the response. Introduce a max_time_waiting for you appl
ication
    waiting_ack = True
    while(waiting_ack):
        recv_ack = lora_sock.recv(256)

        if (len(recv_ack) > 0):
            device_id, pkg_len, ack = struct.unpack(_LORA_PKG_ACK_FORMAT, recv_ack)
            if (device_id == DEVICE_ID):
                if (ack == 200):
                    waiting_ack = False
                    # If the uart = machine.UART(0, 115200) and os.dupterm(uart) are s
et in the boot.py this print should appear in the serial port
                    print("ACK")
                else:
                    waiting_ack = False
                    # If the uart = machine.UART(0, 115200) and os.dupterm(uart) are s
et in the boot.py this print should appear in the serial port
                    print("Message Failed")

    time.sleep(5)
```

The node is always sending packages and waiting for the `ack` from the gateway.

> To adapt this code to user specific needs:
>
> - Put a max waiting time for the `ack` to arrive and resend the package or mark it as invalid
> - Increase the package size changing the `_LORA_PKG_FORMAT` to `BH%ds`. The `H` will allow the keeping of 2 bytes for size (for more information about struct format)
> - Reduce the package size with bitwise manipulation
> - Reduce the message size (for this demo, a string) to something more useful for specific development

LoRa Module to Module Connection

This example shows how to connect two Pycode LoRa capable modules (nodes) via raw LoRa.

Node A

```
from network import LoRa
import socket
import time

# Please pick the region that matches where you are using the device:
# Asia = LoRa.AS923
# Australia = LoRa.AU915
# Europe = LoRa.EU868
# United States = LoRa.US915
lora = LoRa(mode=LoRa.LORA, region=LoRa.EU868)
s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
s.setblocking(False)

while True:
    if s.recv(64) == b'Ping':
        s.send('Pong')
    time.sleep(5)
```

Node B

```
from network import LoRa
import socket
import time

# Please pick the region that matches where you are using the device:
# Asia = LoRa.AS923
# Australia = LoRa.AU915
# Europe = LoRa.EU868
# United States = LoRa.US915
lora = LoRa(mode=LoRa.LORA, region=LoRa.EU868)
s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
s.setblocking(False)
while True:
    s.send('Ping')
    time.sleep(5)
```

LoRaWAN Nano-Gateway

This example allows to connect a LoPy to a LoRaWAN network such as The Things Network (TTN) or Loriot to be used as a nano-gateway.

This example uses settings specifically for connecting to The Things Network within the European 868 MHz region. For another usage, please see the `config.py` file for relevant sections that need changing.

Up to date versions of these snippets can be found at the following GitHub Repository. For more information and discussion about this code, see this forum post.

Nano-Gateway

The Nano-Gateway code is split into 3 files, `main.py`, `config.py` and `nanogateway.py`. These are used to configure and specify how the gateway will connect to a preferred network and how it can act as packet forwarder.

Gateway ID

Most LoRaWAN network servers expect a Gateway ID in the form of a unique 64-bit hexadecimal number (called a EUI-64). The recommended practice is to produce this ID from your board by expanding the WiFi MAC address (a 48-bit number, called MAC-48). You can obtain that by running this code prior to configuration:

```
from network import WLAN
import ubinascii
wl = WLAN()
ubinascii.hexlify(wl.mac())[:6] + 'FFFE' + ubinascii.hexlify(wl.mac())[6:]
```

The result will by something like `b'240ac4FFFE008d88'` where `240ac4FFFE008d88` is your Gateway ID to be used in your network provider configuration.

Main (`main.py`)

This file runs at boot and calls the library and `config.py` files to initialise the nano-gateway. Once configuration is set, the nano-gateway is then started.

```python
""" LoPy LoRaWAN Nano Gateway example usage """

import config
from nanogateway import NanoGateway

if __name__ == '__main__':
    nanogw = NanoGateway(
        id=config.GATEWAY_ID,
        frequency=config.LORA_FREQUENCY,
        datarate=config.LORA_GW_DR,
        ssid=config.WIFI_SSID,
        password=config.WIFI_PASS,
        server=config.SERVER,
        port=config.PORT,
        ntp_server=config.NTP,
        ntp_period=config.NTP_PERIOD_S
        )

    nanogw.start()
    nanogw._log('You may now press ENTER to enter the REPL')
    input()
```

Configuration (`config.py`)

This file contains settings for the server and network it is connecting to. Depending on the nano-gateway region and provider (TTN, Loriot, etc.) these will vary. The provided example will work with The Things Network (TTN) in the European, 868Mhz, region.

The Gateway ID is generated in the script using the process described above.

Please change the WIFI_SSID and WIFI_PASS variables to match your desired WiFi network

```
""" LoPy LoRaWAN Nano Gateway configuration options """

import machine
import ubinascii

WIFI_MAC = ubinascii.hexlify(machine.unique_id()).upper()
# Set  the Gateway ID to be the first 3 bytes of MAC address + 'FFFE' + last 3 bytes o
f MAC address
GATEWAY_ID = WIFI_MAC[:6] + "FFFE" + WIFI_MAC[6:12]

SERVER = 'router.eu.thethings.network'
PORT = 1700

NTP = "pool.ntp.org"
NTP_PERIOD_S = 3600

WIFI_SSID = 'my-wifi'
WIFI_PASS = 'my-wifi-password'

# for EU868
LORA_FREQUENCY = 868100000
LORA_GW_DR = "SF7BW125" # DR_5
LORA_NODE_DR = 5

# for US915
# LORA_FREQUENCY = 903900000
# LORA_GW_DR = "SF7BW125" # DR_3
# LORA_NODE_DR = 3
```

Library (`nanogateway.py`)

The nano-gateway library controls all of the packet generation and forwarding for the LoRa data. This does not require any user configuration and the latest version of this code should be downloaded from the Pycom GitHub Repository.

```
""" LoPy Nano Gateway class """

from network import WLAN
from network import LoRa
from machine import Timer
import os
import ubinascii
import machine
import json
import time
import errno
import _thread
import socket
```

```
PROTOCOL_VERSION = const(2)

PUSH_DATA = const(0)
PUSH_ACK = const(1)
PULL_DATA = const(2)
PULL_ACK = const(4)
PULL_RESP = const(3)

TX_ERR_NONE = "NONE"
TX_ERR_TOO_LATE = "TOO_LATE"
TX_ERR_TOO_EARLY = "TOO_EARLY"
TX_ERR_COLLISION_PACKET = "COLLISION_PACKET"
TX_ERR_COLLISION_BEACON = "COLLISION_BEACON"
TX_ERR_TX_FREQ = "TX_FREQ"
TX_ERR_TX_POWER = "TX_POWER"
TX_ERR_GPS_UNLOCKED = "GPS_UNLOCKED"

STAT_PK = {"stat": {"time": "", "lati": 0,
                   "long": 0, "alti": 0,
                   "rxnb": 0, "rxok": 0,
                   "rxfw": 0, "ackr": 100.0,
                   "dwnb": 0, "txnb": 0}}

RX_PK = {"rxpk": [{"time": "", "tmst": 0,
                  "chan": 0, "rfch": 0,
                  "freq": 868.1, "stat": 1,
                  "modu": "LORA", "datr": "SF7BW125",
                  "codr": "4/5", "rssi": 0,
                  "lsnr": 0, "size": 0,
                  "data": ""}]}

TX_ACK_PK = {"txpk_ack":{"error":""}}


class NanoGateway:

    def __init__(self, id, frequency, datarate, ssid, password, server, port, ntp='poo
l.ntp.org', ntp_period=3600):
            self.id = id
        self.frequency = frequency
        self.sf = self._dr_to_sf(datarate)
        self.ssid = ssid
        self.password = password
        self.server = server
        self.port = port
        self.ntp = ntp
        self.ntp_period = ntp_period

        self.rxnb = 0
        self.rxok = 0
                self.rxfw = 0
                self.dwnb = 0
```

```
                self.txnb = 0

        self.stat_alarm = None
                self.pull_alarm = None
                self.uplink_alarm = None

        self.udp_lock = _thread.allocate_lock()

        self.lora = None
        self.lora_sock = None

    def start(self):
        # Change WiFi to STA mode and connect
        self.wlan = WLAN(mode=WLAN.STA)
        self._connect_to_wifi()

        # Get a time Sync
        self.rtc = machine.RTC()
        self.rtc.ntp_sync(self.ntp, update_period=self.ntp_period)

        # Get the server IP and create an UDP socket
        self.server_ip = socket.getaddrinfo(self.server, self.port)[0][-1]
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UD
P)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.sock.setblocking(False)

        # Push the first time immediately
        self._push_data(self._make_stat_packet())

        # Create the alarms
        self.stat_alarm = Timer.Alarm(handler=lambda t: self._push_data(self._make_sta
t_packet()), s=60, periodic=True)
        self.pull_alarm = Timer.Alarm(handler=lambda u: self._pull_data(), s=25, perio
dic=True)

        # Start the UDP receive thread
        _thread.start_new_thread(self._udp_thread, ())

        # Initialize LoRa in LORA mode
        self.lora = LoRa(mode=LoRa.LORA, frequency=self.frequency, bandwidth=LoRa.BW_1
25KHZ, sf=self.sf,
                        preamble=8, coding_rate=LoRa.CODING_4_5, tx_iq=True)
        # Create a raw LoRa socket
        self.lora_sock = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
        self.lora_sock.setblocking(False)
        self.lora_tx_done = False

        self.lora.callback(trigger=(LoRa.RX_PACKET_EVENT | LoRa.TX_PACKET_EVENT), hand
ler=self._lora_cb)

    def stop(self):
        # TODO: Check how to stop the NTP sync
```

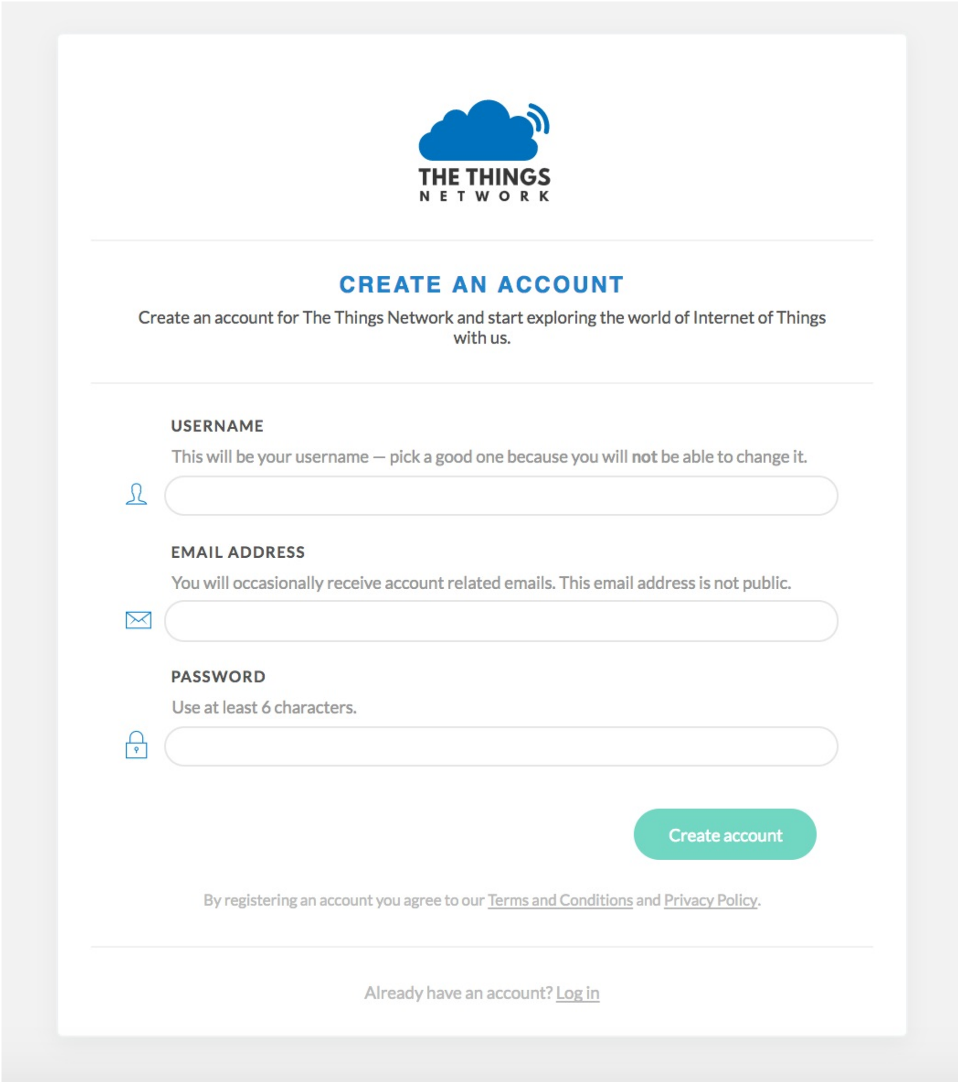
```
        # TODO: Create a cancel method for the alarm
        # TODO: kill the UDP thread
        self.sock.close()

    def _connect_to_wifi(self):
        self.wlan.connect(self.ssid, auth=(None, self.password))
        while not self.wlan.isconnected():
            time.sleep(0.5)
        print("WiFi connected!")

    def _dr_to_sf(self, dr):
        sf = dr[2:4]
        if sf[1] not in '0123456789':
            sf = sf[:1]
        return int(sf)

    def _sf_to_dr(self, sf):
        return "SF7BW125"

    def _make_stat_packet(self):
        now = self.rtc.now()
        STAT_PK["stat"]["time"] = "%d-%02d-%02d %02d:%02d:%02d GMT" % (now[0], now[1],
 now[2], now[3], now[4], now[5])
        STAT_PK["stat"]["rxnb"] = self.rxnb
        STAT_PK["stat"]["rxok"] = self.rxok
        STAT_PK["stat"]["rxfw"] = self.rxfw
        STAT_PK["stat"]["dwnb"] = self.dwnb
        STAT_PK["stat"]["txnb"] = self.txnb
        return json.dumps(STAT_PK)

    def _make_node_packet(self, rx_data, rx_time, tmst, sf, rssi, snr):
        RX_PK["rxpk"][0]["time"] = "%d-%02d-%02dT%02d:%02d:%02d.%dZ" % (rx_time[0], rx
_time[1], rx_time[2], rx_time[3], rx_time[4], rx_time[5], rx_time[6])
        RX_PK["rxpk"][0]["tmst"] = tmst
        RX_PK["rxpk"][0]["datr"] = self._sf_to_dr(sf)
        RX_PK["rxpk"][0]["rssi"] = rssi
        RX_PK["rxpk"][0]["lsnr"] = float(snr)
        RX_PK["rxpk"][0]["data"] = ubinascii.b2a_base64(rx_data)[:-1]
        RX_PK["rxpk"][0]["size"] = len(rx_data)
        return json.dumps(RX_PK)

    def _push_data(self, data):
        token = os.urandom(2)
        packet = bytes([PROTOCOL_VERSION]) + token + bytes([PUSH_DATA]) + ubinascii.un
hexlify(self.id) + data
        with self.udp_lock:
            try:
                self.sock.sendto(packet, self.server_ip)
            except Exception:
                print("PUSH exception")

    def _pull_data(self):
        token = os.urandom(2)
```

```
        packet = bytes([PROTOCOL_VERSION]) + token + bytes([PULL_DATA]) + ubinascii.un
hexlify(self.id)
        with self.udp_lock:
            try:
                self.sock.sendto(packet, self.server_ip)
            except Exception:
                print("PULL exception")

    def _ack_pull_rsp(self, token, error):
        TX_ACK_PK["txpk_ack"]["error"] = error
        resp = json.dumps(TX_ACK_PK)
        packet = bytes([PROTOCOL_VERSION]) + token + bytes([PULL_ACK]) + ubinascii.unh
exlify(self.id) + resp
        with self.udp_lock:
            try:
                self.sock.sendto(packet, self.server_ip)
            except Exception:
                print("PULL RSP ACK exception")

    def _lora_cb(self, lora):
        events = lora.events()
        if events & LoRa.RX_PACKET_EVENT:
            self.rxnb += 1
            self.rxok += 1
            rx_data = self.lora_sock.recv(256)
            stats = lora.stats()
            self._push_data(self._make_node_packet(rx_data, self.rtc.now(), stats.time
stamp, stats.sf, stats.rssi, stats.snr))
            self.rxfw += 1
        if events & LoRa.TX_PACKET_EVENT:
            self.txnb += 1
            lora.init(mode=LoRa.LORA, frequency=self.frequency, bandwidth=LoRa.BW_125K
HZ,
                      sf=self.sf, preamble=8, coding_rate=LoRa.CODING_4_5, tx_iq=True)

    def _send_down_link(self, data, tmst, datarate, frequency):
        self.lora.init(mode=LoRa.LORA, frequency=frequency, bandwidth=LoRa.BW_125KHZ,
                       sf=self._dr_to_sf(datarate), preamble=8, coding_rate=LoRa.CODING
_4_5,
                       tx_iq=True)
        while time.ticks_us() < tmst:
            pass
        self.lora_sock.send(data)

    def _udp_thread(self):
        while True:
            try:
                data, src = self.sock.recvfrom(1024)
                _token = data[1:3]
                _type = data[3]
                if _type == PUSH_ACK:
                    print("Push ack")
                elif _type == PULL_ACK:
```

```
                    print("Pull ack")
                elif _type == PULL_RESP:
                    self.dwnb += 1
                    ack_error = TX_ERR_NONE
                    tx_pk = json.loads(data[4:])
                    tmst = tx_pk["txpk"]["tmst"]
                    t_us = tmst - time.ticks_us() - 5000
                    if t_us < 0:
                        t_us += 0xFFFFFFFF
                    if t_us < 20000000:
                        self.uplink_alarm = Timer.Alarm(handler=lambda x: self._send_d
own_link(ubinascii.a2b_base64(tx_pk["txpk"]["data"]),

        tx_pk["txpk"]["tmst"] - 10, tx_pk["txpk"]["datr"],

        int(tx_pk["txpk"]["freq"] * 1000000)), us=t_us)
                    else:
                        ack_error = TX_ERR_TOO_LATE
                        print("Downlink timestamp error!, t_us:", t_us)
                    self._ack_pull_rsp(_token, ack_error)
                    print("Pull rsp")
            except socket.timeout:
                pass
            except OSError as e:
                if e.errno == errno.EAGAIN:
                    pass
                else:
                    print("UDP recv OSError Exception")
            except Exception:
                print("UDP recv Exception")
            # Wait before trying to receive again
            time.sleep(0.025)
```

Registering with TTN

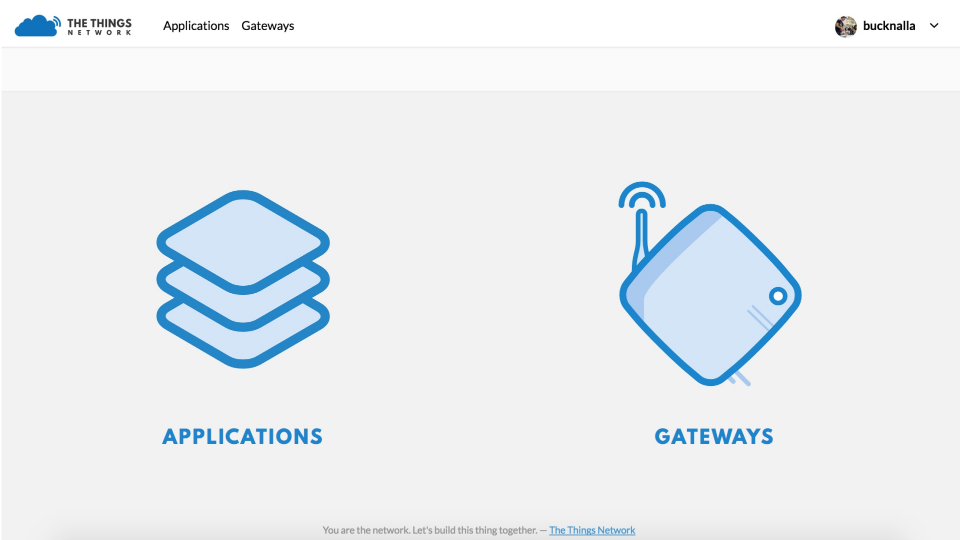
To set up the gateway with The Things Network (TTN), navigate to their website and create/register an account. Enter a username and an email address to verify with their platform.



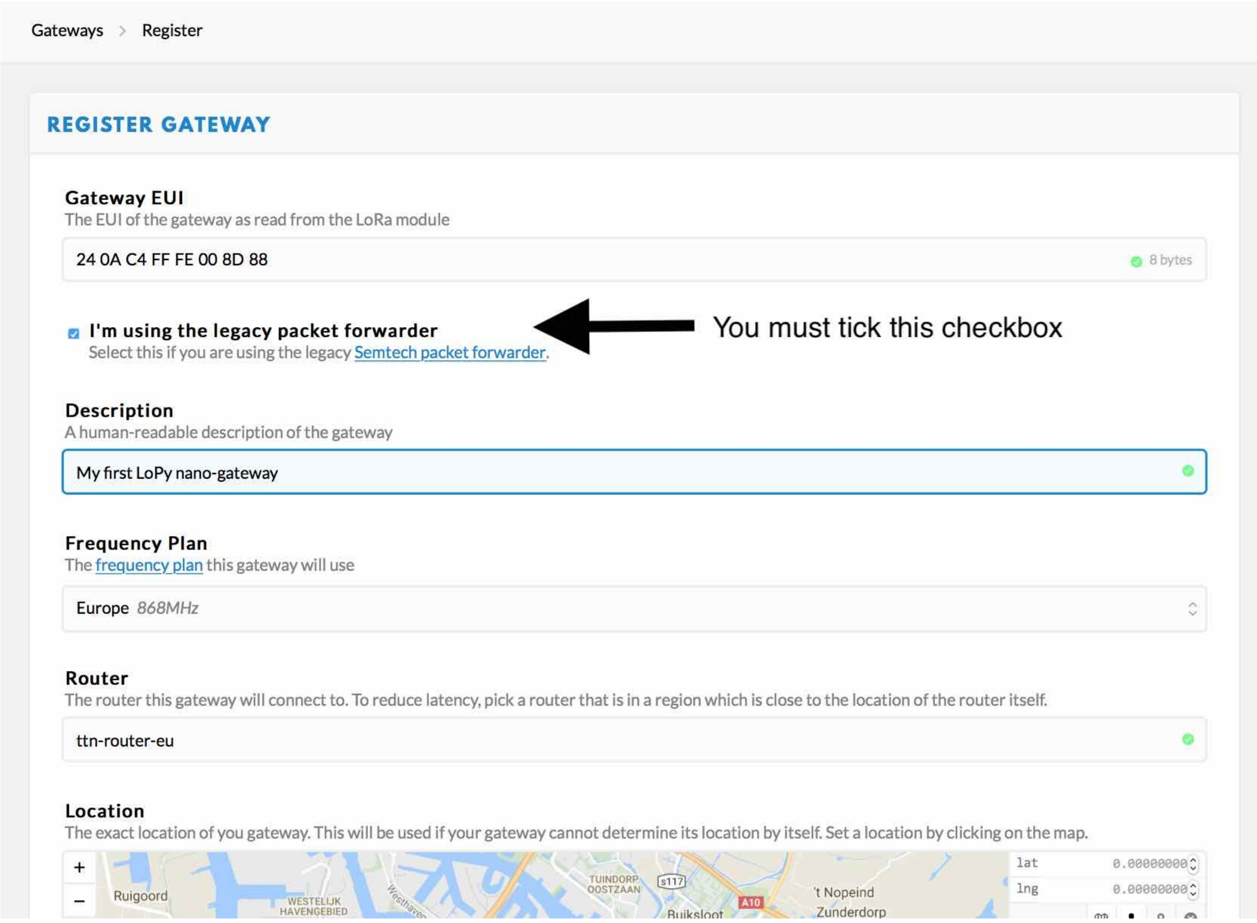
Once an account has been registered, the nano-gateway can then be registered. To do this, navigate to the TTN Console web page.

Registering the Gateway

Inside the TTN Console, there are two options, `applications` and `gateways`. Select `gateways` and then click on `register gateway`. This will allow for the set up and registration of a new nano-gateway.



On the Register Gateway page, you will need to set the following settings:



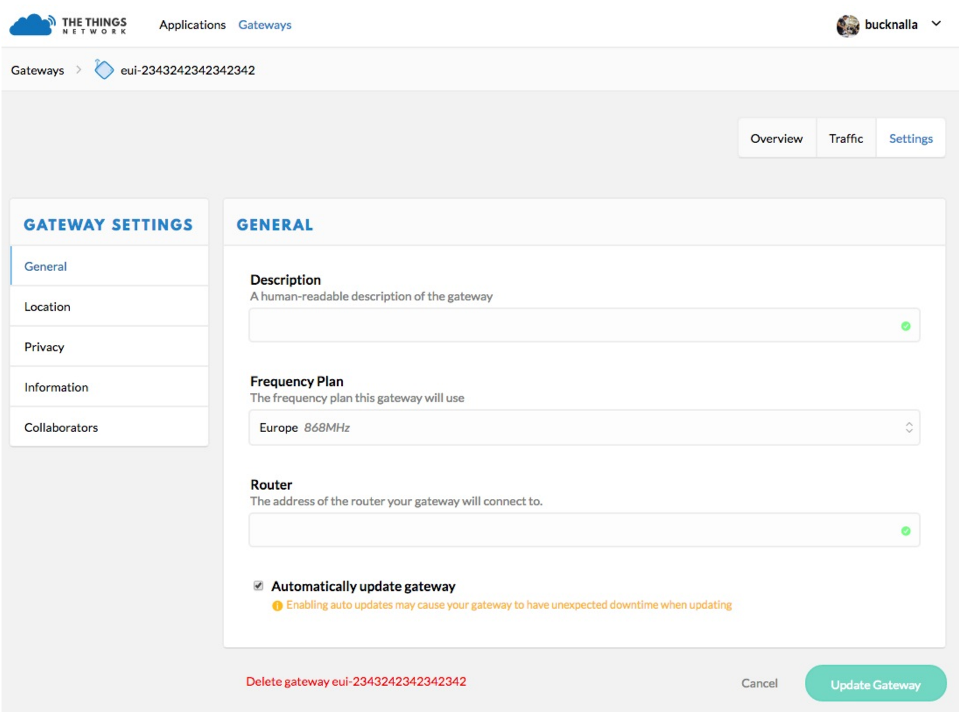
These are unique to each gateway, location and country specific frequency. Please verify that correct settings are selected otherwise the gateway will not connect to TTN.

You need to tick the "I'm using the legacy packet forwarder" to enable the right settings. This is because the Nano-Gateway uses the 'de facto' standard Semtech UDP protocol.

Option	Value
Protocol	Packet Forwarder
Gateway EUI	User Defined (must match `config.py`)
Description	User Defined
Frequency Plan	Select Country (e.g. EU - 868 MHz)
Location	User Defined
Antenna Placement	Indoor or Outdoor

The Gateway EUI should match your Gateway ID from the `config.py` file. We suggest you follow the procedure described near the top of this document to create your own unique Gateway ID.

Once these settings have been applied, click `Register Gateway` . A Gateway Overview page will appear, with the configuration settings showing. Next click on the `Gateway Settings` and configure the Router address to match that of the gateway (default: `router.eu.thethings.network`).



The `Gateway` should now be configured. Next, one or more nodes can now be configured to use the nano-gateway and TTN applications may be built.

LoPy Node

There are two methods of connecting LoPy devices to the nano-gateway, Over the Air Activation (OTAA) and Activation By Personalisation (ABP). The code and instructions for registering these methods are shown below, followed by instruction for how to connect them to an application on TTN.

> It's important that the following code examples (also on GitHub) are used to connect to the nano-gateway as it only supports single channel connections.

OTAA (Over The Air Activation)

When the LoPy connects an application (via TTN) using OTAA, the network configuration is derived automatically during a handshake between the LoPy and network server. Note that the network keys derived using the OTAA methodology are specific to the device and are used to encrypt and verify transmissions at the network level.

```
""" OTAA Node example compatible with the LoPy Nano Gateway """

from network import LoRa
import socket
import ubinascii
import struct
import time

# Initialize LoRa in LORAWAN mode.
lora = LoRa(mode=LoRa.LORAWAN)

# create an OTA authentication params
dev_eui = ubinascii.unhexlify('AABBCCDDEEFF7778') # these settings can be found from T
TN
app_eui = ubinascii.unhexlify('70B3D57EF0003BFD') # these settings can be found from T
TN
app_key = ubinascii.unhexlify('36AB7625FE77776881683B495300FFD6') # these settings can
 be found from TTN

# set the 3 default channels to the same frequency (must be before sending the OTAA jo
in request)
lora.add_channel(0, frequency=868100000, dr_min=0, dr_max=5)
lora.add_channel(1, frequency=868100000, dr_min=0, dr_max=5)
lora.add_channel(2, frequency=868100000, dr_min=0, dr_max=5)

# join a network using OTAA
lora.join(activation=LoRa.OTAA, auth=(dev_eui, app_eui, app_key), timeout=0)

# wait until the module has joined the network
while not lora.has_joined():
    time.sleep(2.5)
```

```
    print('Not joined yet...')

# remove all the non-default channels
for i in range(3, 16):
    lora.remove_channel(i)

# create a LoRa socket
s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)

# set the LoRaWAN data rate
s.setsockopt(socket.SOL_LORA, socket.SO_DR, 5)

# make the socket non-blocking
s.setblocking(False)

time.sleep(5.0)

""" Your own code can be written below! """

for i in range (200):
    s.send(b'PKT #' + bytes([i]))
    time.sleep(4)
    rx = s.recv(256)
    if rx:
        print(rx)
    time.sleep(6)
```

ABP (Activation By Personalisation)

Using ABP join mode requires the user to define the following values and input them into both the LoPy and the TTN Application:

- Device Address
- Application Session Key
- Network Session Key

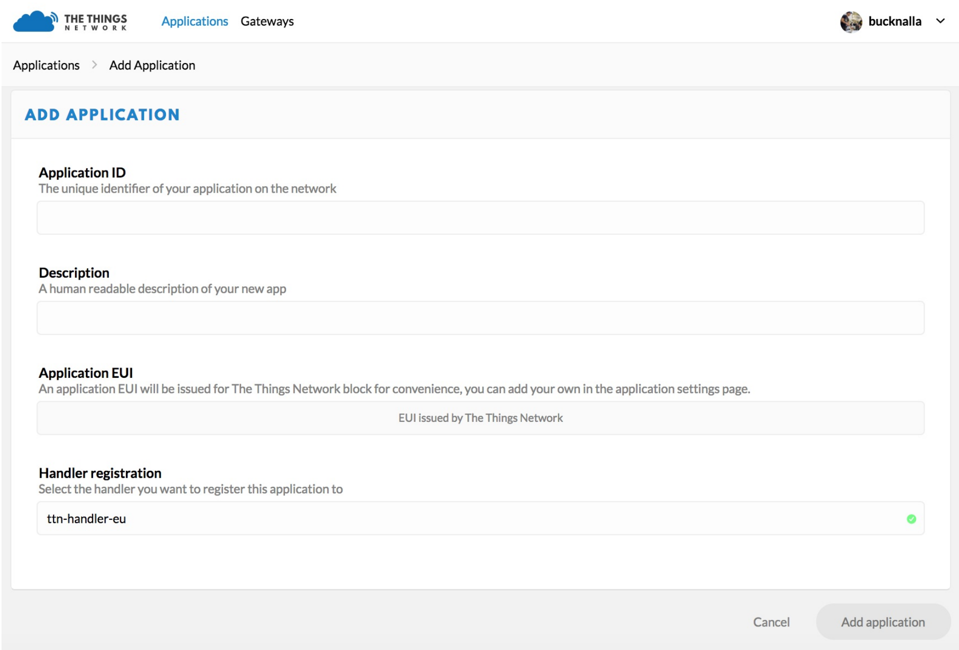
```
""" ABP Node example compatible with the LoPy Nano Gateway """

from network import LoRa
import socket
import ubinascii
import struct
import time

# Initialise LoRa in LORAWAN mode.
lora = LoRa(mode=LoRa.LORAWAN)

# create an ABP authentication params
dev_addr = struct.unpack(">l", ubinascii.unhexlify('2601147D'))[0] # these settings ca
n be found from TTN
nwk_swkey = ubinascii.unhexlify('3C74F4F40CAE2221303BC24284FCF3AF') # these settings c
an be found from TTN
app_swkey = ubinascii.unhexlify('0FFA7072CC6FF69A102A0F39BEB0880F') # these settings c
an be found from TTN

# join a network using ABP (Activation By Personalisation)
lora.join(activation=LoRa.ABP, auth=(dev_addr, nwk_swkey, app_swkey))

# remove all the non-default channels
for i in range(3, 16):
    lora.remove_channel(i)

# set the 3 default channels to the same frequency
lora.add_channel(0, frequency=868100000, dr_min=0, dr_max=5)
lora.add_channel(1, frequency=868100000, dr_min=0, dr_max=5)
lora.add_channel(2, frequency=868100000, dr_min=0, dr_max=5)

# create a LoRa socket
s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)

# set the LoRaWAN data rate
s.setsockopt(socket.SOL_LORA, socket.SO_DR, 5)

# make the socket non-blocking
s.setblocking(False)

""" Your own code can be written below! """

for i in range (200):
    s.send(b'PKT #' + bytes([i]))
    time.sleep(4)
    rx = s.recv(256)
    if rx:
        print(rx)
    time.sleep(6)
```

TTN Applications

Now that the gateway & nodes have been setup, a TTN Application can be built; i.e. what happens to the LoRa data once it is received by TTN. There are a number of different setups/systems that can be used, however the following example demonstrates the HTTP request integration.

Registering an Application

Selecting the `Applications` tab at the top of the TTN console, will bring up a screen for registering applications. Click register and a new page, similar to the one below, will open.



Enter a unique `Application ID` as well as a Description & Handler Registration.

Now the LoPy nodes must be registered to send data up to the new Application.

Registering Devices (LoPy)

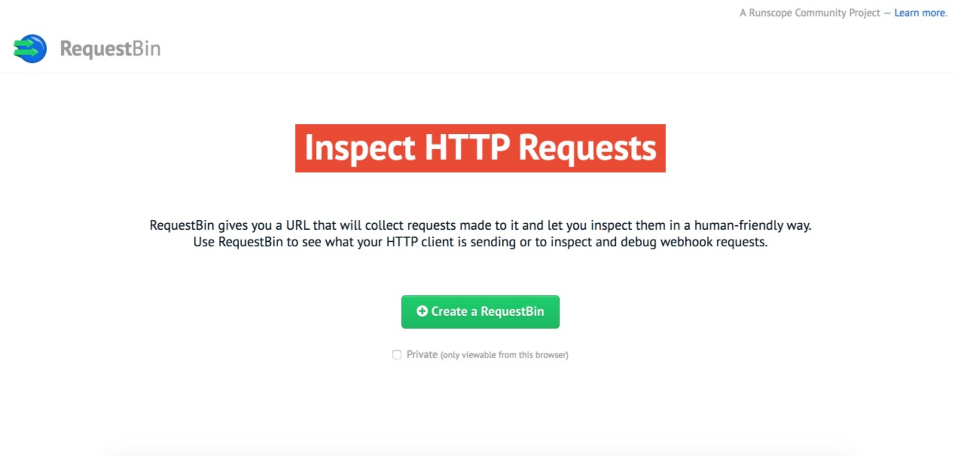
To connect nodes to the nano-gateway, devices need to be added to the application. To do this, navigate to the `Devices` tab on the `Application` home page and click the `Register Device` button.

In the `Register Device` panel, complete the forms for the `Device ID` and the `Device EUI`. The `Device ID` is user specified and is unique to the device in this application. The `Device EUI` is also user specified but must consist of exactly 8 bytes, given in hexadecimal.

Once the device has been added, change the `Activation Method` between `OTAA` and `ABP` depending on user preference. This option can be found under the Settings tab.

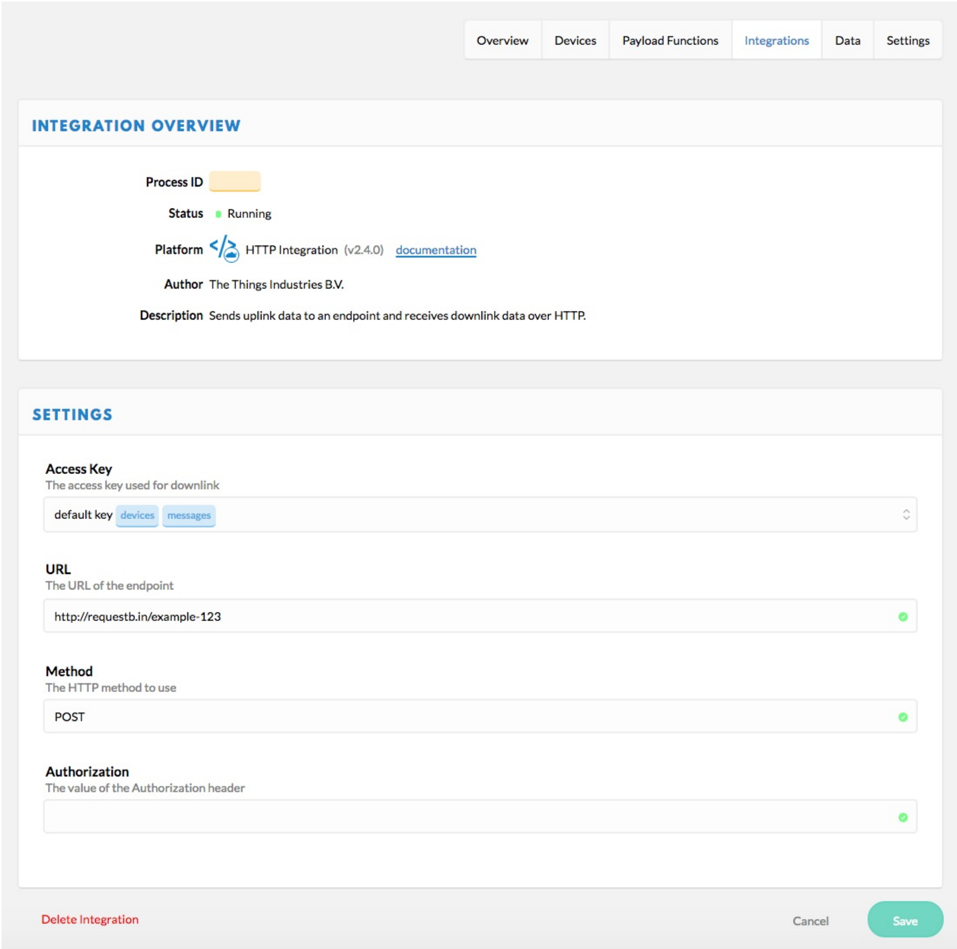
Adding Application Integrations

Now that the data is arriving on the TTN Backend, TTN can be managed as to where data should be delivered to. To do this, use the `Integrations` tab within the new Application's settings.



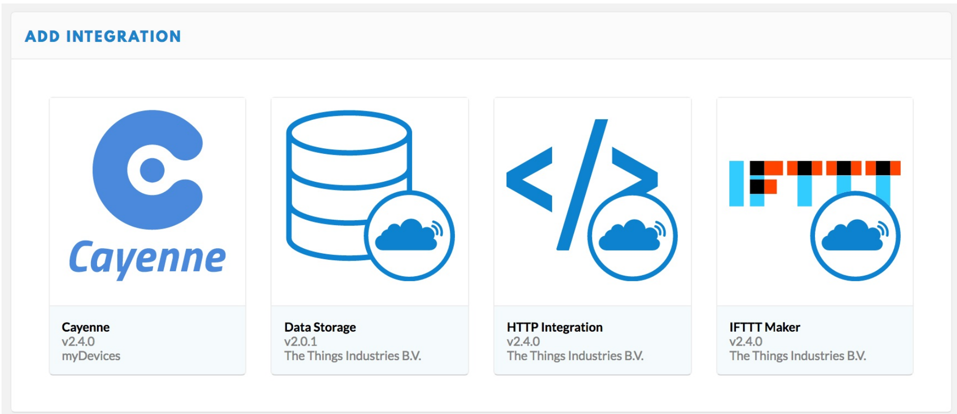
Upon clicking `add integration`, a screen with 4 different options will appear. These have various functionality and more information about them can be found on the TTN website/documentation.

For this example, use the `HTTP Integration` to forward the LoRaWAN Packets to a remote server/address.

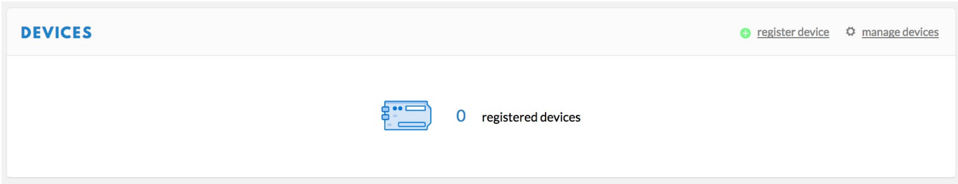


Click `HTTP Integration` to connect up an endpoint that can receive the data.

For testing, a website called RequestBin may be used to receive the data that TTN forwards (via POST Request). To set this up, navigate to RequestBin and click the `Create a RequestBin`.



Copy the URL that is generated and past this into the `URL` form under the `Application Settings`.



This is the address that TTN will forward data onto. As soon as a LoPy starts sending messages, TTN will forward these onto `RequestBin` and they will appear at the unique `RequestBin URL`.

RN2483 to LoPy

This example shows how to send data between a Microchip RN2483 and a LoPy via raw LoRa.

RN2483

```
mac pause
radio set freq 868000000

radio set mod lora
radio set bw 250
radio set sf sf7
radio set cr 4/5
radio set bw 125
radio set sync 12
radio set prlen 8

# Transmit via radio tx:
radio tx 48656c6C6F  #(should send ‘Hello’)
```

LoPy

```
from network import LoRa
import socket

lora = LoRa(mode=LoRa.LORA, frequency= 868000000, bandwidth=LoRa.BW_125KHZ, sf=7, prea
mble=8,
    coding_rate=LoRa.CODING_4_5, power_mode=LoRa.ALWAYS_ON,
    tx_iq=False, rx_iq=False, public=False)

s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)

# This keeps listening for data "forever".
while(True):
    s.recv(64)
```

SiPy Tutorials

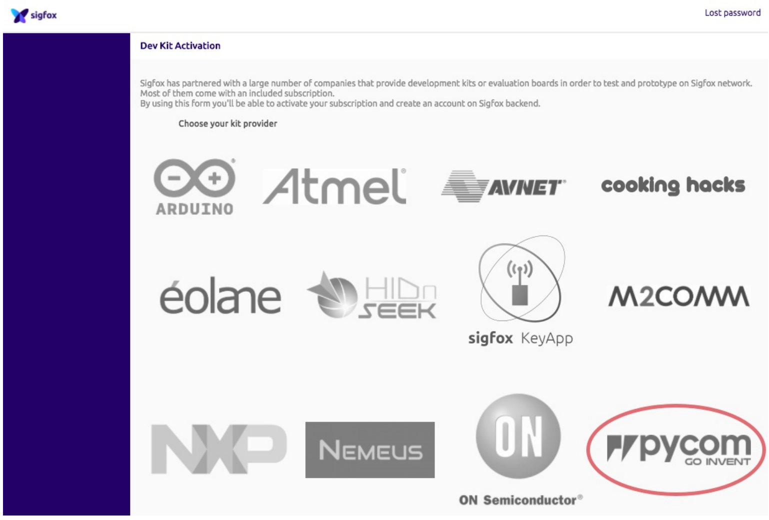
To ensure your device has been provisioned with **Device ID** and **PAC number**, please update to the latest firmware.

The following tutorials demonstrate how to register and get started with the SiPy. The SiPy can be configured for operation in various countries based upon specified RCZ zones (see the `Sigfox` class for more info). The SiPy supports both uplink and downlink `Sigfox` messages as well as device to device communication via its FSK Mode `Sigfox`.

Registering with Sigfox

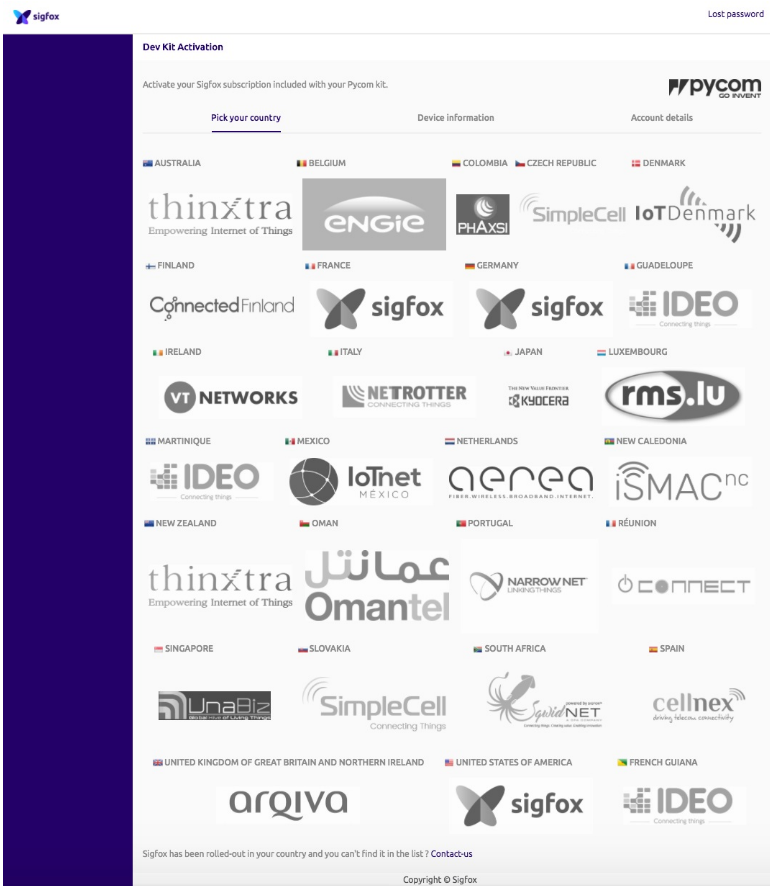
To ensure the device has been provisioned with **Device ID** and **PAC number**, please update to the latest firmware.

In order to send a Sigfox message, the device need to register with the Sigfox Backend. Navigate to https://backend.sigfox.com/activate to find the list of Sigfox enabled development kits.

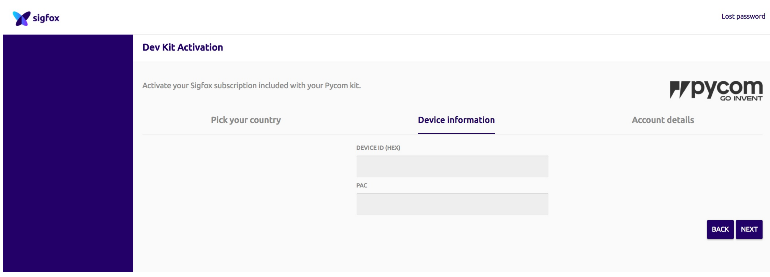


Select `Pycom` to proceed.

Next choose a Sigfox Operator for the country where the device will be activated. Find the specific country and select the operator to continue.



Now need to enter the device's **Device ID** and **PAC number**.

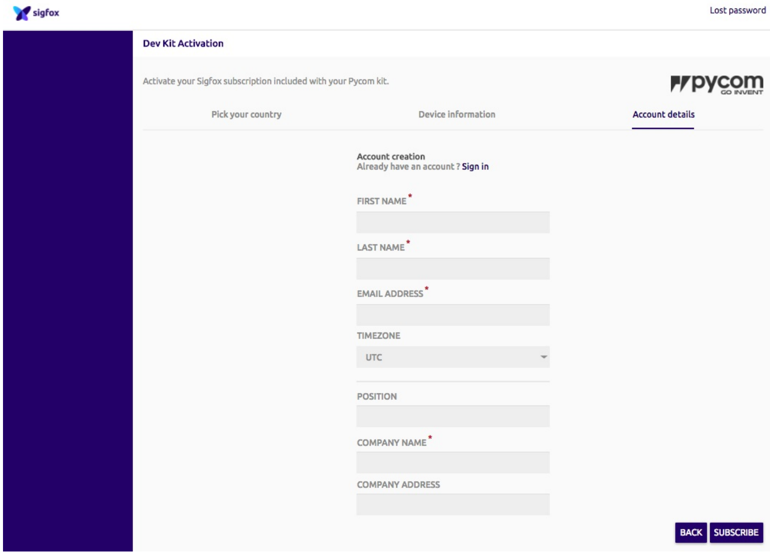


The **Device ID** and **PAC number** are retrievable through a couple of commands via the REPL.

```
from network import Sigfox
import ubinascii

# initalise Sigfox for RCZ1 (You may need a different RCZ Region)
sigfox = Sigfox(mode=Sigfox.SIGFOX, rcz=Sigfox.RCZ1)

# print Sigfox Device ID
print(ubinascii.hexlify(sigfox.id()))

# print Sigfox PAC number
print(ubinascii.hexlify(sigfox.pac()))
```

See `Sigfox` for more info about the Sigfox Class and which `RCZ` region to use.

Once the device's Device ID and PAC number have been entered, create an account. Provide the required information including email address and click to continue.



An email confirming the creation of a Sigfox Backend account and the successful registration of the device should arrive at the users inbox.

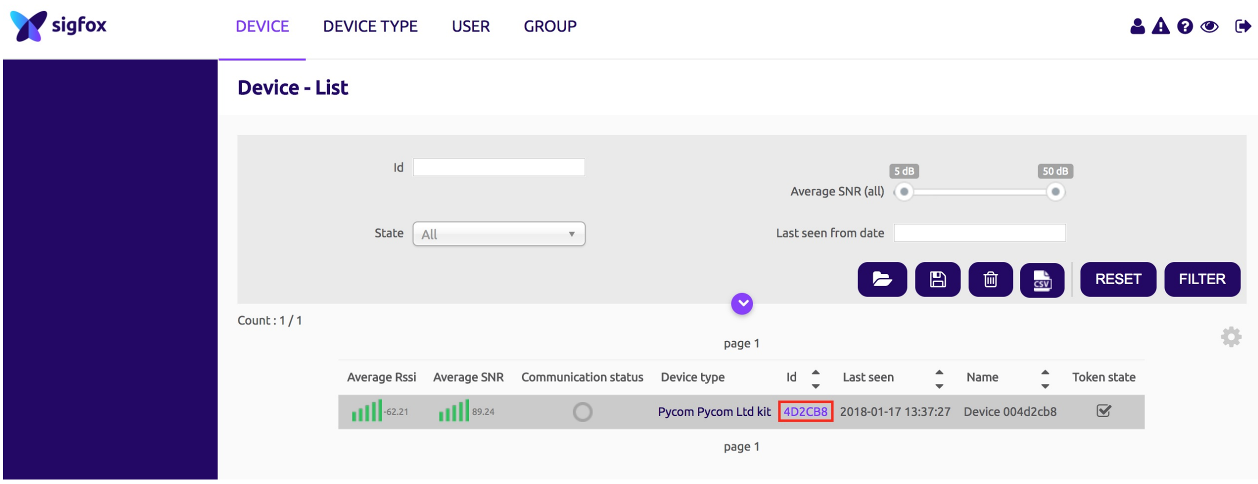
How To Disengage Sequence Number

If your are experiencing issues with Sigfox connectivity, this could be due to the sequence number being out of sync. To prevent replay on the network, the Sigfox protocol uses sequence numbers. If there is a large difference between the sequence number sent by the device and the one expected by the backend, your message is dropped by the network.

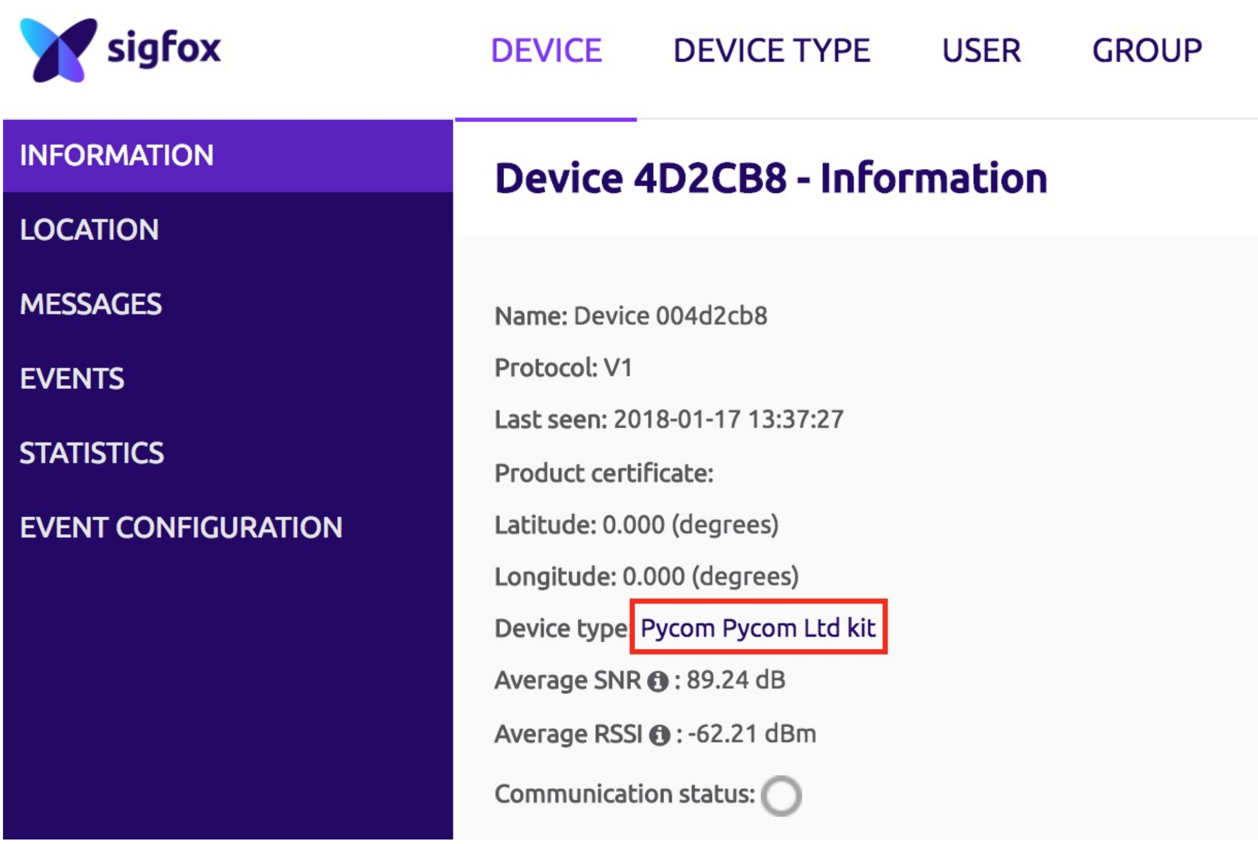
You can use the `Disengage sequence number` button on the device information or on the device type information page of the Sigfox backend to reset the number expected by the backend. If the sequence number of your next message is different from the last trashed sequence number, the message will be accepted.

Issues with the sequence number can occur when a lot of messages are sent when outside of Sigfox coverage for instance.

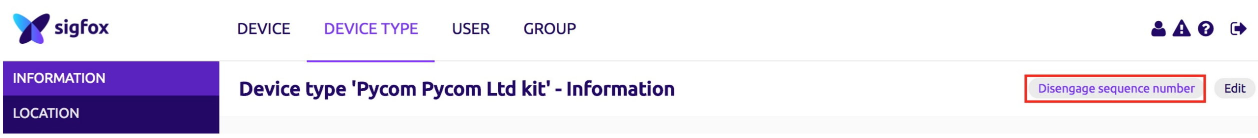
Firstly you will need to log into the Sigfox Backend, navigate to device, and click on the Sigfox ID of the affected SiPy.



You should now see the Information page with an entry `Device Type:` followed by a link. Please follow the link



Finally, on this page click on `Disengage sequence number` button in the upper right corner.



LTE Tutorials

The following tutorials demonstrate the use of the LTE CAT-M1 and NB-IoT functionality on cellular enabled Pycom modules.

Our cellular modules support both LTE CAT-M1 and NB-IoT, these are new lower power, long range, cellular protocols. These are not the same as the full version of 2G/3G/LTE supported by cell phones, and require your local carriers to support them. At the time of writing, CAT-M1 and NB-IoT connectivity is not widely available so be sure to check with local carriers if support is available where you are.

LTE class for Cat M1

Please ensure you have the latest Sequans modem firmware for the best network compatibility. Instructions for this can be found here.

The LTE Cat M1 service gives full IP access through the cellular modem.

Once the `lte.connect()` function has completed all the IP socket functions - including SSL - will be routed through this connection. This mean any code using WLAN can be adapted to Cat M1 by simply adding the connection setup step first and disconnect after.

For example to connect over LTE Cat M1 to Google's web server over secure SSL:

```
import socket
import ssl
import time
from network import LTE

lte = LTE()         # instantiate the LTE object
lte.attach()        # attach the cellular modem to a base station
while not lte.isattached():
    time.sleep(0.25)
lte.connect()       # start a data session and obtain an IP address
while not lte.isconnected():
    time.sleep(0.25)

s = socket.socket()
s = ssl.wrap_socket(s)
s.connect(socket.getaddrinfo('www.google.com', 443)[0][-1])
s.send(b"GET / HTTP/1.0\r\n\r\n")
print(s.recv(4096))
s.close()

lte.disconnect()
lte.dettach()
```

This also applies to our MQTT and AWS examples.

IMPORTANT: Once the LTE radio is initialised, it must be de-initialised before going to deepsleep in order to ensure minimum power consumption. This is required due to the LTE radio being powered independently and allowing use cases which require the system to be taken out from deepsleep by an event from the LTE network (data or SMS received for instance).

When using the expansion board and the FiPy together, the RTS/CTS jumpers **MUST** be removed as those pins are being used by the LTE radio. Keeping those jumpers in place will lead to erratic operation and higher current consumption specially while in deepsleep.

LTE class for Narrow Band IoT

As shipped, Pycom modules only support CAT-M1, in order to use NB-IoT you need to flash a different firmware to the Sequans modem. Instructions for this can be found here.

Current NB-IoT limitations

At the moment the NB-IoT firmware supplied by Sequans only support Ericsson base stations configured for In-Band mode. Standalone and guard-band modes will be supported in a later release. Support for Huawei base stations is also limited and only lab testing with Huawei eNodeB is recommended at the moment. Full support for Huawei is planned for early Q2 2018.

NB-IoT usage:

Example with Vodafone:

```
from network import LTE

lte = LTE()
lte.send_at_cmd('AT+CFUN=0')
lte.send_at_cmd('AT!="clearscanconfig"')
lte.send_at_cmd('AT!="addscanfreq band=20 dl-earfcn=6300"')
lte.send_at_cmd('AT!="zsp0:npc 1"')
lte.send_at_cmd('AT+CGDCONT=1,"IP","nb.inetd.gdsp"')
lte.send_at_cmd('AT+CFUN=1')

while not lte.isattached():
    pass

lte.connect()
while not lte.isconnected():
    pass

# now use socket as usual...
```

IMPORTANT: Once the LTE radio is initialised, it must be de-initialised before going to deepsleep in order to ensure minimum power consumption. This is required due to the LTE radio being powered independently and allowing use cases which require the system to be taken out from deepsleep by an event from the LTE network (data or SMS received for instance).

When using the expansion board and the FiPy together, the RTS/CTS jumpers **MUST** be removed as those pins are being used by the LTE radio. Keeping those jumpers in place will lead to erratic operation and higher current consumption specially while in deepsleep.

How to get the IMEI of your module

In order to retrieve the IMEI of your cellular enabled Pycom module you will firstly need to make sure you are on firmware version `1.17.0.b1` or higher. You can check your firmware version by running the following code on you device via the interactive REPL.

```
>>> import os
>>> os.uname()
(sysname='GPy', nodename='GPy', release='1.17.0.b1', version='v1.8.6-849-d0dc708 on 20
18-02-27', machine='GPy with ESP32')
```

Once you have a compatible firmware, you can run the following code to get your modules IMEI number:

```
from network import LTE
lte = LTE()
lte.send_at_cmd('AT+CGSN=1')
```

You'll get a return string like this `\r\n+CGSN: "354347xxxxxxxxx"\r\n\r\nOK`. The value between the double quotes is your IMEI.

Firmware upgrade tool for the Sequans Monarch SQN3330

Description

The Sequans Monarch SQN3330 cellular radio found on the Pycom FiPy, GPy and GO1 modules requires a different firmware to operate in CAT-M1 or NB-IoT mode.

This page will explain the process to upgrade the firmware of the cellular radio The process involves streaming the firmware file from the ESP32 to the SQN3330. Currently, the file has to be stored in a micro SD card first so that the ESP32 can access it easily. We are current working to add support for streaming the file via the updater tool as well.

Requirements

Before proceeding you will need:

- Pycom cellular enabled module (GPy, FiPy, G01)
- FAT32 formatted microSD card (with at least 6MB of free space)
- A Pycom Expansion Board or shield (or a microSD card socket breakout board)

Usage

If your module is running the factory LTE chip firmware, you **MUST** first perform an update to the latest CAT-M1 firmware before trying to upgrade to the NB-IoT firmware. Skipping this step will cause your radio to become unresponsive and it will require access to the test points in order to re-flash the firmware.

Firstly, you will need to download the required library files from here. You will need to place these in a directory called "lib" just like any other libraries. This can be done using either FTP or Pymakr

Next you need to download the firmware file from here. You will need to place the firmware on a FAT32 formatted microSD card, then insert the SD card into a Expansion Board, Pytrack, Pysense or Pyscan. Power-up the system and connect to the interactive REPL and run the following code:

```
import sqnsupgrade
sqnsupgrade.run(path_to_firmware, 921600)   # path_to_firmware example: '/sd/FIPY_NB1_
35351.dup'
```

The whole process can take between 2 and 3 minutes and at some points it will seem to stall, this is normal, just be patience. You should see an output like this:

```
<<< Welcome to the SQN3330 firmware updater >>>
Entering recovery mode
Resetting.

Starting STP (DO NOT DISCONNECT POWER!!!)
STP started
Session opened: version 1, max transfer 8192 bytes
Sending 4560505 bytes: [########################################] 100%
Code download done, returning to user mode
Resetting (DO NOT DISCONNECT POWER!!!).
.........
Deploying the upgrade (DO NOT DISCONNECT POWER!!!)...
Resetting (DO NOT DISCONNECT POWER!!!)..
...
Upgrade completed!
Here is the current firmware version:
UE6.0.0.0-ER7
LR6.0.0.0-35351
OK
```

DO NOT disconnect power while the upgrade process is taking place, wait for it to finish!

If the module get's stuck in here for more than 1 minute while upgrading to the NB-IoT firmware, you can cycle power and retry. In this case it is safe.

```
Sending 4560505 bytes: [##                                      ]   6%
```

Accelerometer

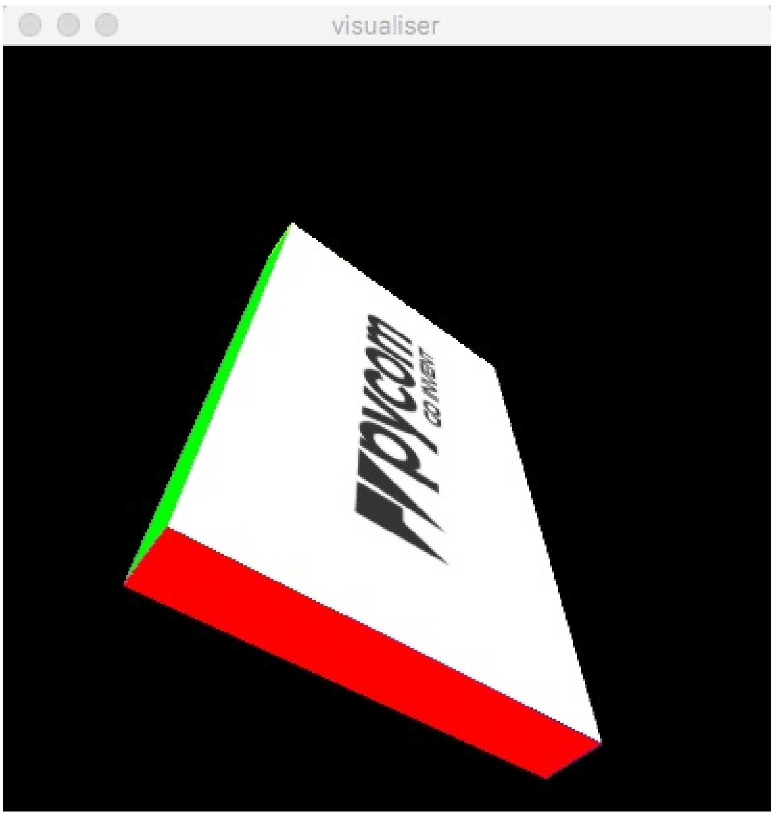
Both the Pysense and Pytrack use the same accelerometer. Please see the Pysense Examples to see how to use the accelerometer.

Sensor Demos

Accelerometer

This basic example shows how to read pitch and roll from the on-board accelerometer and output it in comma separated value (CSV) format over serial.

```
from LIS2HH12 import LIS2HH12
from pytrack import Pytrack
py = Pytrack()
acc = LIS2HH12()

while True:
    pitch = acc.pitch()
    roll = acc.roll()
    print('{},{}'.format(pitch, roll))
    time.sleep_ms(100)
```



If you want to visualise the data output by this script a Processing sketch is available here that will show the board orientation in 3D.

Introduction

This chapter describes modules (function and class libraries) that are built into MicroPython. There are a number of categories for the available modules:

- Modules which implement a subset of standard Python functionality and are not intended to be extended by the user.
- Modules which implement a subset of Python functionality, with a provision for extension by the user (via Python code).
- Modules which implement MicroPython extensions to the Python standard libraries.
- Modules specific to a particular port and thus not portable.

Note about the availability of modules and their contents

This documentation in general aspires to describe all modules and functions/classes which are implemented in MicroPython. However, MicroPython is highly configurable, and each port to a particular board/embedded system makes available only a subset of MicroPython libraries. For officially supported ports, there is an effort to either filter out non-applicable items, or mark individual descriptions with “Availability:” clauses describing which ports provide a given feature. With that in mind, please still be warned that some functions/classes in a module (or even the entire module) described in this documentation may be unavailable in a particular build of MicroPython on a particular board. The best place to find general information of the availability/non-availability of a particular feature is the “General Information” section which contains information pertaining to a specific port.

Beyond the built-in libraries described in this documentation, many more modules from the Python standard library, as well as further MicroPython extensions to it, can be found in the micropython-lib repository.

Pycom Modules

These modules are specific to the Pycom devices and may have slightly different implementations to other variations of MicroPython (i.e. for Non-Pycom devices). Modules include those which support access to underlying hardware, e.g. I2C, SPI, WLAN, Bluetooth, etc.

module machine

The `machine` module contains specific functions related to the board.

Quick Usage Example

```
import machine

help(machine) # display all members from the machine module
machine.freq() # get the CPU frequency
machine.unique_id() # return the 6-byte unique id of the board (the LoPy's WiFi MAC ad
dress)
```

Reset Functions

machine.reset()

Resets the device in a manner similar to pushing the external RESET button.

machine.reset_cause()

Get the reset cause. See constants for the possible return values.

Interrupt Functions

machine.disable_irq()

Disable interrupt requests. Returns and integer representing the previous IRQ state. This return value can be passed to `enable_irq` to restore the IRQ to its original state.

machine.enable_irq([state])

Enable interrupt requests. The most common use of this function is to pass the value returned by `disable_irq` to exit a critical section. Another options is to enable all interrupts which can be achieved by calling the function with no parameters.

Power Functions

machine.freq()

Returns CPU frequency in hertz.

machine.idle()

Gates the clock to the CPU, useful to reduce power consumption at any time during short or long periods. Peripherals continue working and execution resumes as soon as any interrupt is triggered (on many ports this includes system timer interrupt occurring at regular intervals on the order of millisecond).

machine.deepsleep([time_ms])

Stops the CPU and all peripherals, including the networking interfaces (except for LTE). Execution is resumed from the main script, just as with a reset. If a value in milliseconds is given then the device will wake up after that period of time, otherwise it will remain in deep sleep until the reset button is pressed.

The products with LTE connectivity (FiPy, GPy, G01), require the LTE radio to be disabled separately via the LTE class before entering deepsleep. This is required due to the LTE radio being powered independently and allowing use cases which require the system to be taken out from deepsleep by an event from the LTE network (data or SMS received for instance).

machine.pin_deepsleep_wakeup(pins, mode, enable_pull)

Configure pins to wake up from deep sleep mode. The pins which have this capability are: `P2, P3, P4, P6, P8 to P10 and P13 to P23`.

The arguments are:

- `pins` a list or tuple containing the `GPIO` to setup for deepsleep wakeup.
- `mode` selects the way the configure `GPIO`s can wake up the module. The possible values are: `machine.WAKEUP_ALL_LOW` and `machine.WAKEUP_ANY_HIGH`.
- `enable_pull` if set to `True` keeps the pull up or pull down resistors enabled during deep sleep. If this variable is set to `True`, then `ULP` or capacitive touch wakeup cannot be used in combination with `GPIO` wakeup.

machine.wake_reason()

Get the wake reason. See constants for the possible return values. Returns a tuple of the form: `(wake_reason, gpio_list)`. When the wakeup reason is either GPIO or touch pad, then the second element of the tuple is a list with GPIOs that generated the wakeup.

machine.remaining_sleep_time()

Returns the remaining timer duration (in milliseconds) if the ESP32 is woken up from deep sleep by something other than the timer. For example, if you set the timer for 30 seconds (30000 ms) and it wakes up after 10 seconds then this function will return `20000`.

Miscellaneous Functions

machine.main(filename)

Set the `filename` of the main script to run after `boot.py` is finished. If this function is not called then the default file `main.py` will be executed.

It only makes sense to call this function from within `boot.py`.

machine.rng()

Return a 24-bit software generated random number.

machine.unique_id()

Returns a byte string with a unique identifier of a board/SoC. It will vary from a board/SoC instance to another, if underlying hardware allows. Length varies by hardware (so use substring of a full value if you expect a short ID). In some MicroPython ports, ID corresponds to the network MAC address.

> Use `ubinascii.hexlify()` to convert the byte string to hexadecimal form for ease of manipulation and use elsewhere.

machine.info()

Returns the high water mark of the stack associated with various system tasks, in words (1 word = 4 bytes on the ESP32). If the value is zero then the task has likely overflowed its stack. If the value is close to zero then the task has come close to overflowing its stack.

Constants

Reset Causes

machine.PWRON_RESET machine.HARD_RESET machine.WDT_RESET machine.DEEPSLEEP_RESET machine.SOFT_RESET machine.BROWN_OUT_RESET

Wake Reasons

machine.PWRON_WAKE machine.PIN_WAKE machine.RTC_WAKE machine.ULP_WAKE

Pin Wakeup Modes

machine.WAKEUP_ALL_LOW machine.WAKEUP_ANY_HIGH

class ADC – Analog to Digital Conversion

Quick Usage Example

```
import machine

adc = machine.ADC()             # create an ADC object
apin = adc.channel(pin='P16')   # create an analog pin on P16
val = apin()                    # read an analog value
```

Constructors

class machine.ADC(id=0)

Create an ADC object; associate a channel with a pin. For more info check the hardware section.

Methods

adc.init(* , bits=12)

Enable the ADC block. This method is automatically called on object creation.

- `Bits` can take values between 9 and 12 and selects the number of bits of resolution of the ADC block.

adc.deinit()

Disable the ADC block.

adc.channel(* , pin, attn=ADC.ATTN_0DB)

Create an analog pin.

- `pin` is a keyword-only string argument. Valid pins are `P13` to `P20` .
- `attn` is the attenuation level. The supported values are: ADC.ATTN_0DB ADC.ATTN_2_5DB ADC.ATTN_6DB ADC.ATTN_11DB

Returns an instance of ADCChannel. Example:

```
# enable an ADC channel on P16
apin = adc.channel(pin='P16')
```

adc.vref(vref)

If called without any arguments, this function returns the current calibrated voltage (in millivolts) of the `1.1v` reference. Otherwise it will update the calibrated value (in millivolts) of the internal `1.1v` reference.

adc.vref_to_pin(pin)

Connects the internal `1.1v` to external `GPIO`. It can only be connected to `P22`, `P21` or `P6`. It is recommended to only use `P6` on the WiPy, on other modules this pin is connected to the radio.

Constants

ADC.ATTN_0DB ADC.ATTN_2_5DB ADC.ATTN_6DB ADC.ATTN_11DB
ADC channel attenuation values

class ADCChannel

Read analog values from internal/external sources. ADC channels can be connected to internal points of the `MCU` or to `GPIO` pins. ADC channels are created using the `ADC.channel` method.

Methods

adcchannel()

Fast method to read the channel value.

adcchannel.value()

Read the channel value.

adcchannel.init()

(Re)init and enable the ADC channel. This method is automatically called on object creation.

adcchannel.deinit()

Disable the ADC channel.

adcchannel.voltage()

Reads the channels value and converts it into a voltage (in millivolts)

adcchannel.value_to_voltage(value)

Converts the provided value into a voltage (in millivolts) in the same way voltage does.

ADC pin input range is `0-1.1V`. This maximum value can be increased up to `3.3V` using the highest attenuation of `11dB`. **Do not exceed the maximum of 3.3V**, to avoid damaging the device.

class DAC – Digital to Analog Conversion

The DAC is used to output analog values (a specific voltage) on pin `P22` or pin `P21` . The voltage will be between `0` and `3.3V` .

Quick Usage Example

```
import machine

dac = machine.DAC('P22')        # create a DAC object
dac.write(0.5)                  # set output to 50%

dac_tone = machine.DAC('P21')   # create a DAC object
dac_tone.tone(1000, 0)          # set tone output to 1kHz
```

Constructors

class class machine.DAC(pin)

Create a DAC object, that will let you associate a channel with a `pin` . `pin` can be a string argument.

Methods

dac.init()

Enable the DAC block. This method is automatically called on object creation.

dac.deinit()

Disable the DAC block.

dac.write(value)

Set the DC level for a DAC pin. `value` is a float argument, with values between 0 and 1.

dac.tone(frequency, amplitude)

Sets up tone signal to the specified `frequency` at `amplitude` scale. `frequency` can be from `125Hz` to `20kHz` in steps of `122Hz` . `amplitude` is an integer specifying the tone amplitude to write the DAC pin. Amplitude value represents:

- `0` is 0dBV (~ 3Vpp at 600 Ohm load)
- `1` is -6dBV (~1.5 Vpp), `2` is -12dBV (~0.8 Vpp)
- `3` is -18dBV (~0.4 Vpp). The generated signal is a sine wave with an DC offset of VDD/2.

class I2C – Two-Wire Serial Protocol

I2C is a two-wire protocol for communicating between devices. At the physical level it consists of 2 wires: SCL and SDA, the clock and data lines respectively.

I2C objects are created attached to a specific bus. They can be initialised when created, or initialised later on.

Example using default Pins

```
from machine import I2C

i2c = I2C(0)                         # create on bus 0
i2c = I2C(0, I2C.MASTER)             # create and init as a master
i2c = I2C(0, pins=('P10','P11'))     # create and use non-default PIN assignments (P10
=SDA, P11=SCL)
i2c.init(I2C.MASTER, baudrate=20000) # init as a master
i2c.deinit()                         # turn off the peripheral
```

Example using non-default Pins

```
from machine import I2C

i2c = I2C(0, pins=('P10','P11'))     # create and use non-default PIN assignments (P10
=SDA, P11=SCL)
i2c.init(I2C.MASTER, baudrate=20000) # init as a master
i2c.deinit()                         # turn off the peripheral
```

Printing the `i2c` object gives you information about its configuration.

A master must specify the recipient's address:

```
i2c.init(I2C.MASTER)
i2c.writeto(0x42, '123')         # send 3 bytes to slave with address 0x42
i2c.writeto(addr=0x42, b'456')   # keyword for address
```

Master also has other methods:

```
i2c.scan()                        # scan for slaves on the bus, returning
                                  #   a list of valid addresses
i2c.readfrom_mem(0x42, 2, 3)      # read 3 bytes from memory of slave 0x42,
                                  #   starting at address 2 in the slave
i2c.writeto_mem(0x42, 2, 'abc')   # write 'abc' (3 bytes) to memory of slave 0x42
                                  # starting at address 2 in the slave, timeout afte
r 1 second
```

Quick Usage Example

```
from machine import I2C
# configure the I2C bus
i2c = I2C(0, I2C.MASTER, baudrate=100000)
i2c.scan() # returns list of slave addresses
i2c.writeto(0x42, 'hello') # send 5 bytes to slave with address 0x42
i2c.readfrom(0x42, 5) # receive 5 bytes from slave
i2c.readfrom_mem(0x42, 0x10, 2) # read 2 bytes from slave 0x42, slave memory 0x10
i2c.writeto_mem(0x42, 0x10, 'xy') # write 2 bytes to slave 0x42, slave memory 0x10
```

Constructors

class machine.I2C(bus, ...)

Construct an I2C object on the given `bus`. `bus` can only be `0, 1, 2`. If the `bus` is not given, the default one will be selected (`0`). Buses `0` and `1` use the ESP32 I2C hardware peripheral while bus `2` is implemented with a bit-banged software driver.

General Methods

i2c.init(mode, * , baudrate=100000, pins=(SDA, SCL))

Initialise the I2C bus with the given parameters:

- `mode` must be I2C.MASTER
- `baudrate` is the SCL clock rate
- pins is an optional tuple with the pins to assign to the I2C bus. The default I2C pins are `P9` (SDA) and `P10` (SCL)

i2c.scan()

Scan all I2C addresses between `0x08` and `0x77` inclusive and return a list of those that respond. A device responds if it pulls the SDA line low after its address (including a read bit) is sent on the bus.

Standard Bus Operations

The following methods implement the standard I2C master read and write operations that target a given slave device.

i2c.readfrom(addr, nbytes)

Read `nbytes` from the slave specified by `addr`. Returns a bytes object with the data read.

i2c.readfrom_into(addr, buf)

Read into `buf` from the slave specified by `addr`. The number of bytes read will be the length of `buf`.

Return value is the number of bytes read.

i2c.writeto(addr, buf, * , stop=True)

Write the bytes from `buf` to the slave specified by `addr`. The argument `buf` can also be an integer which will be treated as a single byte. If `stop` is set to `False` then the stop condition won't be sent and the I2C operation may be continued (typically with a read transaction).

Return value is the number of bytes written.

Memory Operations

Some I2C devices act as a memory device (or set of registers) that can be read from and written to. In this case there are two addresses associated with an I2C transaction: the slave address and the memory address. The following methods are convenience functions to communicate with such devices.

i2c.readfrom_mem(addr, memaddr, nbytes, *, addrsize=8)

Read `nbytes` from the slave specified by `addr` starting from the memory address specified by `memaddr`. The `addrsize` argument is specified in bits and it can only take 8 or 16.

i2c.readfrom_mem_into(addr, memaddr, buf, *, addrsize=8)

Read into `buf` from the slave specified by `addr` starting from the memory address specified by `memaddr`. The number of bytes read is the length of `buf`. The `addrsize` argument is specified in bits and it can only take 8 or 16.

The return value is the number of bytes read.

i2c.writeto_mem(addr, memaddr, buf *, addrsize=8)

Write `buf` to the slave specified by `addr` starting from the memory address specified by `memaddr`. The argument `buf` can also be an integer which will be treated as a single byte. The `addrsize` argument is specified in bits and it can only take 8 or 16.

The return value is the number of bytes written.

Constants

I2C.MASTER
Used to initialise the bus to master mode.

class Pin – Control I/O Pins

A pin is the basic object to control I/O pins (also known as GPIO - general-purpose input/output). It has methods to set the mode of the pin (input, output, etc) and methods to get and set the digital logic level. For analog control of a pin, see the ADC class.

Quick Usage Example

```
from machine import Pin

# initialize `P9` in gpio mode and make it an output
p_out = Pin('P9', mode=Pin.OUT)
p_out.value(1)
p_out.value(0)
p_out.toggle()
p_out(True)

# make `P10` an input with the pull-up enabled
p_in = Pin('P10', mode=Pin.IN, pull=Pin.PULL_UP)
p_in() # get value, 0 or 1
```

Constructors

class machine.Pin(id, ...)

Create a new Pin object associated with the string `id`. If additional arguments are given, they are used to initialise the pin. See pin.init().

```
from machine import Pin
p = Pin('P10', mode=Pin.OUT, pull=None, alt=-1)
```

Methods

pin.init(mode, pull, * , alt)

Initialise the pin:

- `mode` can be one of:
  - Pin.IN - input pin.
  - Pin.OUT - output pin in push-pull mode.
  - Pin.OPEN_DRAIN - input or output pin in open-drain mode.

- `pull` can be one of:
  - `None` - no pull up or down resistor.
  - Pin.PULL_UP - pull up resistor enabled.
  - Pin.PULL_DOWN - pull down resistor enabled.
- `alt` is the id of the alternate function.

Returns: `None` .

pin.id()

Get the pin id.

pin.value([value])

Get or set the digital logic level of the pin:

- With no argument, return 0 or 1 depending on the logic level of the pin.
- With value given, set the logic level of the pin. value can be anything that converts to a boolean. If it converts to True, the pin is set high, otherwise it is set low.

pin([value])

Pin objects are callable. The call method provides a (fast) shortcut to set and get the value of the pin.

Example:

```
from machine import Pin
pin = Pin('P12', mode=Pin.IN, pull=Pin.PULL_UP)
pin()   # fast method to get the value
```

See pin.value() for more details.

pin.toggle()

Toggle the value of the pin.

pin.mode([mode])

Get or set the pin mode.

pin.pull([pull])

Get or set the pin pull.

pin.hold([hold])

Get or set the pin hold. You can apply a hold to a pin by passing `True` (or clear it by passing `False`). When a pin is held, its value cannot be changed by using `Pin.value()` or `Pin.toggle()` until the hold is released. This Can be used to retain the pin state through a core reset and system reset triggered by watchdog time-out or Deep-sleep events. Only pins in the RTC power domain can retain their value through deep sleep or reset. These are: `P2, P3, P4, P6, P8, P9, P10, P13, P14, P15, P16, P17, P18, P19, P20, P21, P22, P23`.

pin.callback(trigger, handler=None, arg=None)

Set a callback to be triggered when the input level at the pin changes.

- `trigger` is the type of event that triggers the callback. Possible values are:
  - Pin.IRQ_FALLING interrupt on falling edge.
  - Pin.IRQ_RISING interrupt on rising edge.
  - Pin.IRQ_LOW_LEVEL interrupt on low level.
  - Pin.IRQ_HIGH_LEVEL interrupt on high level.

The values can be OR-ed together, for instance `trigger=Pin.IRQ_FALLING | Pin.IRQ_RISING`

- `handler` is the function to be called when the event happens. This function will receive one argument. Set `handler` to `None` to disable it.
- `arg` is an optional argument to pass to the callback. If left empty or set to `None`, the function will receive the Pin object that triggered it.

Example:

```
from machine import Pin

def pin_handler(arg):
    print("got an interrupt in pin %s" % (arg.id()))

p_in = Pin('P10', mode=Pin.IN, pull=Pin.PULL_UP)
p_in.callback(Pin.IRQ_FALLING | Pin.IRQ_RISING, pin_handler)
```

For more information on how Pycom's products handle interrupts, see here.

Attributes

***class* pin.exp_board**

Contains all Pin objects supported by the expansion board. Examples:

```
Pin.exp_board.G16
led = Pin(Pin.exp_board.G16, mode=Pin.OUT)
Pin.exp_board.G16.id()
```

***class* pin.module**

Contains all `Pin` objects supported by the module. Examples:

```
Pin.module.P9
led = Pin(Pin.module.P9, mode=Pin.OUT)
Pin.module.P9.id()
```

Constants

The following constants are used to configure the pin objects. Note that not all constants are available on all ports.

Pin.IN Pin.OUT Pin.OPEN_DRAIN
Selects the pin mode.

Pin.PULL_UP Pin.PULL_DOWN
Enables the pull up or pull down resistor.

class PWM – Pulse Width Modulation

Quick Usage Example

```
from machine import PWM
pwm = PWM(0, frequency=5000)  # use PWM timer 0, with a frequency of 5KHz
# create pwm channel on pin P12 with a duty cycle of 50%
pwm_c = pwm.channel(0, pin='P12', duty_cycle=0.5)
pwm_c.duty_cycle(0.3) # change the duty cycle to 30%
```

Constructors

class machine.PWM(timer, frequency)

Create a PWM object. This sets up the `timer` to oscillate at the specified `frequency`. `timer` is an integer from 0 to 3. `frequency` is an integer from 1 Hz to 78 KHz (this values can change in future upgrades).

Methods

pwm.channel(id, pin * , duty_cycle=0.5)

Connect a PWM channel to a pin, setting the initial duty cycle. `id` is an integer from 0 to 7. `pin` is a string argument. `duty_cycle` is a keyword-only float argument, with values between 0 and 1. Returns an instance of `PWMChannel`.

class PWMChannel — PWM channel

Methods

pwmchannel.duty_cycle(value)

Set the duty cycle for a PWM channel. `value` is a float argument, with values between 0 and 1.

class RTC – Real Time Clock

The RTC is used to keep track of the date and time.

Quick Usage Example

```
from machine import RTC

rtc = RTC()
rtc.init((2014, 5, 1, 4, 13, 0, 0, 0))
print(rtc.now())
```

Constructors

class machine.RTC(id=0, ...)

Create an RTC object. See init for parameters of initialisation.

```
# id of the RTC may be set if multiple are connected. Defaults to id = 0.
rtc = RTC(id=0)
```

Methods

rtc.init(datetime=None, source=RTC.INTERNAL_RC)

Initialise the RTC. The arguments are:

- `datetime` when passed it sets the current time. It is a tuple of the form: `(year, month, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]])` .
- `source` selects the oscillator that drives the RTC. The options are RTC.INTERNAL_RC and RTC.XTAL_32KHZ

For example:

```
# for 2nd of February 2017 at 10:30am (TZ 0)
rtc.init((2017, 2, 28, 10, 30, 0, 0, 0))
```

`tzinfo` is ignored by this method. Use `time.timezone` to achieve similar results.

rtc.now()

Get get the current `datetime` tuple:

```
# returns datetime tuple
rtc.now()
```

rtc.ntp_sync(server, * , update_period=3600)

Set up automatic fetch and update the time using NTP (SNTP).

- `server` is the URL of the NTP server. Can be set to `None` to disable the periodic updates.
- `update_period` is the number of seconds between updates. Shortest period is 15 seconds.

Can be used like:

```
rtc.ntp_sync("pool.ntp.org") # this is an example. You can select a more specific serv
er according to your geographical location
```

rtc.synced()

Returns `True` if the last `ntp_sync` has been completed, `False` otherwise:

```
rtc.synced()
```

Constants

RTC.INTERNAL_RC RTC.XTAL_32KHZ
Clock source

class SPI – Serial Peripheral Interface

SPI is a serial protocol that is driven by a master. At the physical level there are 3 lines: SCK, MOSI, MISO.

See usage model of I2C; SPI is very similar. Main difference is parameters to init the SPI bus:

```
from machine import SPI
spi = SPI(0, mode=SPI.MASTER, baudrate=1000000, polarity=0, phase=0, firstbit=SPI.MSB)
```

Only required parameter is mode, must be SPI.MASTER. Polarity can be 0 or 1, and is the level the idle clock line sits at. Phase can be 0 or 1 to sample data on the first or second clock edge respectively.

Quick Usage Example

```
from machine import SPI

# configure the SPI master @ 2MHz
# this uses the SPI default pins for CLK, MOSI and MISO (``P10``, ``P11`` and ``P14``)
spi = SPI(0, mode=SPI.MASTER, baudrate=2000000, polarity=0, phase=0)
spi.write(bytes([0x01, 0x02, 0x03, 0x04, 0x05])) # send 5 bytes on the bus
spi.read(5) # receive 5 bytes on the bus
rbuf = bytearray(5)
spi.write_readinto(bytes([0x01, 0x02, 0x03, 0x04, 0x05]), rbuf) # send a receive 5 byt
es
```

Quick Usage Example using non-default pins

```
from machine import SPI

# configure the SPI master @ 2MHz
# this uses the SPI non-default pins for CLK, MOSI and MISO (``P19``, ``P20`` and ``P2
1``)
spi = SPI(0, mode=SPI.MASTER, baudrate=2000000, polarity=0, phase=0, pins=('P19','P20',
'P21'))
spi.write(bytes([0x01, 0x02, 0x03, 0x04, 0x05])) # send 5 bytes on the bus
spi.read(5) # receive 5 bytes on the bus
rbuf = bytearray(5)
spi.write_readinto(bytes([0x01, 0x02, 0x03, 0x04, 0x05]), rbuf) # send a receive 5 byt
es
```

Constructors

class machine.SPI(id, ...)

Construct an SPI object on the given bus. `id` can be only 0. With no additional parameters, the SPI object is created but not initialised (it has the settings from the last initialisation of the bus, if any). If extra arguments are given, the bus is initialised. See init for parameters of initialisation.

Methods

spi.init(mode, baudrate=1000000, * , polarity=0, phase=0, bits=8, firstbit=SPI.MSB, pins=(CLK, MOSI, MISO))

Initialise the SPI bus with the given parameters:

- `mode` must be SPI.MASTER.
- `baudrate` is the SCK clock rate.
- `polarity` can be 0 or 1, and is the level the idle clock line sits at.
- `phase` can be 0 or 1 to sample data on the first or second clock edge respectively.
- `bits` is the width of each transfer, accepted values are 8, 16 and 32.
- `firstbit` can be SPI.MSB or SPI.LSB.
- `pins` is an optional tuple with the pins to assign to the SPI bus. If the pins argument is not given the default pins will be selected (`P10` as CLK, `P11` as MOSI and `P14` as MISO). If pins is passed as None then no pin assignment will be made.

spi.deinit()

Turn off the SPI bus.

spi.write(buf)

Write the data contained in `buf`. Returns the number of bytes written.

spi.read(nbytes, * , write=0x00)

Read the `nbytes` while writing the data specified by `write`. Return the number of bytes read.

spi.readinto(buf, * , write=0x00)

Read into the buffer specified by `buf` while writing the data specified by `write`. Return the number of bytes read.

spi.write_readinto(write_buf, read_buf)

Write from `write_buf` and read into `read_buf`. Both buffers must have the same length. Returns the number of bytes written

Constants

SPI.MASTER
For initialising the SPI bus to master

SPI.MSB
Set the first bit to be the most significant bit

SPI.LSB
Set the first bit to be the least significant bit

class UART – Universal Asynchronous Receiver/Transmitter

UART implements the standard UART/USART duplex serial communications protocol. At the physical level it consists of 2 lines: RXD and TXD. The unit of communication is a character (not to be confused with a string character) which can be 5, 6, 7 or 8 bits wide.

UART objects can be created and initialised using:

```
from machine import UART

uart = UART(1, 9600)                         # init with given baudrate
uart.init(9600, bits=8, parity=None, stop=1) # init with given parameters
```

Bits can be `5, 6, 7, 8`. Parity can be `None`, UART.EVEN or UART.ODD. Stop can be `1, 1.5 or 2`.

A UART object acts like a stream object therefore reading and writing is done using the standard stream methods:

```
uart.read(10)       # read 10 characters, returns a bytes object
uart.readall()      # read all available characters
uart.readline()     # read a line
uart.readinto(buf)  # read and store into the given buffer
uart.write('abc')   # write the 3 characters
```

To check if there is anything to be read, use:

```
uart.any()               # returns the number of characters available for reading
```

Quick Usage Example

```
from machine import UART
# this uses the UART_1 default pins for TXD and RXD (``P3`` and ``P4``)
uart = UART(1, baudrate=9600)
uart.write('hello')
uart.read(5) # read up to 5 bytes
```

Quick Usage Example using non-default pins (TXD/RXD only)

```
from machine import UART
# this uses the UART_1 non-default pins for TXD and RXD (``P20`` and ``P21``)
uart = UART(1, baudrate=9600, pins=('P20','P21'))
uart.write('hello')
uart.read(5) # read up to 5 bytes
```

Quick Usage Example using non-default pins (TXD/RXD and flow control)

```
from machine import UART
# this uses the UART_1 non-default pins for TXD, RXD, RTS and CTS (``P20``, ``P21``, `
`P22``and ``P23``)
uart = UART(1, baudrate=9600, pins=('P20', 'P21', 'P22', 'P23'))
uart.write('hello')
uart.read(5) # read up to 5 bytes
```

Constructors

class machine.UART(bus, ...)

Construct a UART object on the given `bus`. `bus` can be `0, 1 or 2`. If the `bus` is not given, the default one will be selected (`0`) or the selection will be made based on the given pins.

On the GPy/FiPy UART2 is unavailable because it is used to communicate with the cellular radio.

Methods

uart.init(baudrate=9600, bits=8, parity=None, stop=1, * , timeout_chars=2, pins=(TXD, RXD, RTS, CTS))

Initialise the UART bus with the given parameters:

- `baudrate` is the clock rate.
- `bits` is the number of bits per character. Can be `5, 6, 7 or 8`.
- `parity` is the parity, `None`, UART.EVEN or UART.ODD.
- `stop` is the number of stop bits, `1 or 2`.
- `timeout_chars` Rx timeout defined in number of characters. The value given here will be multiplied by the time a characters takes to be transmitted at the configured `baudrate`.

- `pins` is a 4 or 2 item list indicating the TXD, RXD, RTS and CTS pins (in that order). Any of the pins can be `None` if one wants the UART to operate with limited functionality. If the RTS pin is given the the RX pin must be given as well. The same applies to CTS. When no pins are given, then the default set of TXD (P1) and RXD (P0) pins is taken, and hardware flow control will be disabled. If `pins=None`, no pin assignment will be made.

uart.deinit()

Turn off the UART bus.

uart.any()

Return the number of characters available for reading.

uart.read([nbytes])

Read characters. If `nbytes` is specified then read at most that many bytes.

Return value: a bytes object containing the bytes read in. Returns `None` on timeout.

uart.readall()

Read as much data as possible.

Return value: a bytes object or `None` on timeout.

uart.readinto(buf[, nbytes])

Read bytes into the `buf`. If `nbytes` is specified then read at most that many bytes. Otherwise, read at most `len(buf)` bytes.

Return value: number of bytes read and stored into `buf` or `None` on timeout.

uart.readline()

Read a line, ending in a newline character. If such a line exists, return is immediate. If the timeout elapses, all available data is returned regardless of whether a newline exists.

Return value: the line read or `None` on timeout if no data is available.

uart.write(buf)

Write the buffer of bytes to the bus.

Return value: number of bytes written or None on timeout.

uart.sendbreak()

Send a break condition on the bus. This drives the bus low for a duration of 13 bits. Return value: `None`.

uart.wait_tx_done(timeout_ms)

Waits at most `timeout_ms` for the last Tx transaction to complete. Returns `True` if all data has been sent and the TX buffer has no data in it, otherwise returns `False`.

Constants

UART.EVEN UART.ODD
Parity types (along with `None`)

UART.RX_ANY
IRQ trigger sources

class WDT – Watchdog Timer

The WDT is used to restart the system when the application crashes and ends up into a non recoverable state. After enabling, the application must "feed" the watchdog periodically to prevent it from expiring and resetting the system.

Quick Usage Example

```
from machine import WDT
wdt = WDT(timeout=2000)  # enable it with a timeout of 2 seconds
wdt.feed()
```

Constructors

class machine.WDT(id=0, timeout)

Create a WDT object and start it. The `id` can only be `0`. See the init method for the parameters of initialisation.

Methods

wdt.init(timeout)

Initialises the watchdog timer. The timeout must be given in milliseconds. Once it is running the WDT cannot be stopped but the timeout can be re-configured at any point in time.

wdt.feed()

Feed the WDT to prevent it from resetting the system. The application should place this call in a sensible place ensuring that the WDT is only fed after verifying that everything is functioning correctly.

class Timer – Measure Time and Set Alarms

Timers can be used for a great variety of tasks, like measuring time spans or being notified that a specific interval has elapsed.

These two concepts are grouped into two different subclasses:

`Chrono` : used to measure time spans. `Alarm` : to get interrupted after a specific interval.

> You can create as many of these objects as needed.

Constructors

class Timer.Chrono()

Create a chronometer object.

class Timer.Alarm(handler=None, s, * , ms, us, arg=None, periodic=False)

Create an Alarm object.

- `handler` : will be called after the interval has elapsed. If set to `None` , the alarm will be disabled after creation.
- `arg` : an optional argument can be passed to the callback handler function. If `None` is specified, the function will receive the object that triggered the alarm.
- `s, ms, us` : the interval can be specified in seconds (float), miliseconds (integer) or microseconds (integer). Only one at a time can be specified.
- `periodic` : an alarm can be set to trigger repeatedly by setting this parameter to `True` .

Methods

Timer.sleep_us()

Delay for a given number of microseconds, should be positive or 0 (for speed, the condition is not enforced). Internally it uses the same timer as the other elements of the `Timer` class. It compensates for the calling overhead, so for example, 100us should be really close to 100us. For times bigger than 10,000us it releases the GIL to let other threads run, so exactitude is not guaranteed for delays longer than that.

class Chrono

Can be used to measure time spans.

Methods

chrono.start()

Start the chronometer.

chrono.stop()

Stop the chronometer.

chrono.reset()

Reset the time count to 0.

chrono.read()

Get the elapsed time in seconds.

chrono.read_ms()

Get the elapsed time in milliseconds.

chrono.read_us()

Get the elapsed time in microseconds.

Example:

```
from machine import Timer
import time

chrono = Timer.Chrono()

chrono.start()
time.sleep(1.25) # simulate the first lap took 1.25 seconds
lap = chrono.read() # read elapsed time without stopping
time.sleep(1.5)
chrono.stop()
total = chrono.read()

print()
print("\nthe racer took %f seconds to finish the race" % total)
print("  %f seconds in the first lap" % lap)
print("  %f seconds in the last lap" % (total - lap))
class Alarm – get interrupted after a specific interval
```

Methods

alarm.callback(handler, * , arg=None)

Specify a callback handler for the alarm. If set to `None`, the alarm will be disabled.

An optional argument `arg` can be passed to the callback handler function. If `None` is specified, the function will receive the object that triggered the alarm.

alarm.cancel()

Disables the alarm.

Example:

```
from machine import Timer

class Clock:

    def __init__(self):
        self.seconds = 0
        self.__alarm = Timer.Alarm(self._seconds_handler, 1, periodic=True)

    def _seconds_handler(self, alarm):
        self.seconds += 1
        print("%02d seconds have passed" % self.seconds)
        if self.seconds == 10:
            alarm.cancel() # stop counting after 10 seconds

clock = Clock()
```

For more information on how Pycom’s products handle interrupts, see notes.

class SD – Secure digital Memory Card

The SD card class allows to configure and enable the memory card module of your Pycom module and automatically mount it as `/sd` as part of the file system. There is a single pin combination that can be used for the SD card, and the current implementation only works in 1-bit mode. The pin connections are as follows:

`P8: DAT0` , `P23: SCLK` and `P4: CMD` (no external pull-up resistors are needed)

If you have one of the Pycom expansion boards, then simply insert the card into the micro SD socket and run your script.

> Make sure your SD card is formatted either as FAT16 or FAT32.

Quick Example Usage:

```
from machine import SD
import os

sd = SD()
os.mount(sd, '/sd')

# check the content
os.listdir('/sd')

# try some standard file operations
f = open('/sd/test.txt', 'w')
f.write('Testing SD card write operations')
f.close()
f = open('/sd/test.txt', 'r')
f.readall()
f.close()
```

Constructors

class machine.SD(id, ...)

Create a SD card object. See sd.init() for parameters if initialisation.

Methods

sd.init(id=0)

Enable the SD card.

sd.deinit()

Disable the SD card.

Please note that the SD card library currently supports FAT16/32 formatted SD cards up to 32 GB. Future firmware updates will increase compatibility with additional formats and sizes.

class CAN – Controller Area Network

The CAN class supports the full CAN 2.0 specification with standard and extended frames, as well as acceptance filtering.

The ESP32 has a built-in CAN controller, but the transceiver needs to be added externally. A recommended device is the SN65HVD230.

Quick Usage Example

```
from machine import CAN

can = CAN(mode=CAN.NORMAL, baudrate=500000, pins=('P22', 'P23'))
can.send(id=12, data=bytes([1, 2, 3, 4, 5, 6, 7, 8]))
can.recv()
```

Constructors

***class* machine.CAN(bus=0, ...)**

Create an CAN object. See init for parameters of initialisation.:

```
# only 1 CAN peripheral is available, so the bus must always be 0
can = CAN(0, mode=CAN.NORMAL, baudrate=500000, pins=('P22', 'P23'))    # pin order is
Tx, Rx
```

Methods

can.init(mode=CAN.NORMAL, baudrate=500000, *, frame_format=CAN.FORMAT_STD, rx_queue_len=128, pins=('P22', 'P23'))

Initialize the CAN controller. The arguments are:

- `mode` can take either CAN.NORMAL or CAN.SILENT. Silent mode is useful for sniffing the bus.
- `baudrate` sets up the bus speed. Acceptable values are between 1 and 1000000.
- `frame_format` defines the frame format to be accepted by the receiver. Useful for filtering frames based on the identifier length. Can tale either CAN.FORMAT_STD or CAN.FORMAT_EXT or CAN.FORMAT_BOTH. If CAN.FORMAT_STD is selected, extended frames won't be received and vice-versa.

- `rx_queue_len` defines the number of messages than can be queued by the receiver. Due to CAN being a high traffic bus, large values are recommended (>= 128), otherwise messages will be dropped specially when no filtering is applied.
- `pins` selects the `Tx` and `Rx` pins (in that order).

can.deinit()

Disables the CAN bus.

```
# disable the CAN bus
can.deinit()
```

can.send(id, * , data=None, rtr=False, extended=False)

Send a CAN frame on the bus

- `id` is the identifier of the message.
- `data` can take up to 8 bytes. It must be left empty is the message to be sent is a remote request (rtr=True).
- `rtr` set it to false to send a remote request.
- `extnted` specifies if the message identifier width should be 11bit (standard) or 29bit (extended).

Can be used like:

```
can.send(id=0x0020, data=bytes([0x01, 0x02, 0x03, 0x04, 0x05]), extended=True)   # sen
ds 5 bytes with an extended identifier

can.send(id=0x010, data=bytes([0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08])) # sen
ds 8 bytes with an standard identifier

can.send(id=0x012, rtr=True)         # sends a remote request for message id=0x12
```

can.recv(timeout=0)

Get a message from the receive queue, and optionally specify a timeout value in **s** (can be a floating point value e.g. `0.2`). This function returns `None` if no messages available. If a message is present, it will be returned as a named tuple with the following form:

`(id, data, rtr, extended)`

```
>>> can.recv()
(id=0x012, data=b'123', rtr=False, extended=False)
```

can.soft_filter(mode, filter_list)

Specify a software filter accepting only the messages that pass the filter test.

There are 3 possible filter modes:

- CAN.FILTER_LIST allows to pass the list of IDs that should be accepted.
- CAN.FILTER_RANGE allows to pass a list or tuple of ID ranges that should be accepted.
- CAN.FILTER_MASK allows to pass a list of tuples of the form: `(filer, mask)` .

With software filters all messages in the bus are received by the CAN controller but only the matching ones are passed to the RX queue. This means that the queue won't be filled up with non relevant messages, but the interrupt overhead will remain as normal. The `filter_list` can contain up to 32 elements.

For example:

```
can.soft_filter(CAN.FILTER_LIST, [0x100, 0x200, 0x300, 0x400])  # only accept identifiers from 0x100, 0x200, 0x300 and 0x400

can.soft_filter(CAN.FILTER_RANGE, [(0x001, 0x010), (0x020, 0x030), (0x040, 0x050)])  # only accept identifiers from 0x001 to 0x010, from 0x020 to 0x030 and from 0x040 to 0x050.

can.soft_filter(CAN.FILTER_MASK, [(0x100, 0x7FF), (0x200, 0x7FC)]) # more of the classic Filter and Mask method.

can.soft_filter(None)   # disable soft filters, all messages are accepted
```

can.callback(trigger, handler=None, arg=None)

Set a callback to be triggered when any of this 3 events are present:

- trigger is the type of event that triggers the callback. Possible values are:
  - CAN.RX_FRAME interrupt whenever a new frame is received.
  - CAN.RX_FIFO_NOT_EMPTY interrupt when a frame is received on an empty FIFO.
  - CAN.RX_FIFO_OVERRUN interrupt when a message is received and the FIFO is full.

The values can be OR-ed together, for instance trigger=CAN.RX_FRAME | CAN.RX_FIFO_OVERRUN

- handler is the function to be called when the event happens. This function will receive one argument. Set handler to None to disable the callback.

- arg is an optional argument to pass to the callback. If left empty or set to None, the function will receive the CAN object that triggered it.

It can be used like this:

```
from machine import CAN

can = CAN(mode=CAN.NORMAL, baudrate=500000, pins=('P22', 'P23'))

def can_cb(can_o):
    print('CAN Rx:', can_o.recv())

can.callback(handler=can_cb, trigger=CAN.RX_FRAME)
```

can.events()

This method returns a value with bits sets (if any) indicating the events that have occurred in the bus. Please note that by calling this function the internal events registry is cleared automatically, therefore calling it immediately for a second time will most likely return a value of 0.

Constants

CAN.NORMAL CAN.SILENT CAN.FORMAT_STD CAN.FORMAT_EXT CAN.FORMAT_BOTH CAN.RX_FRAME CAN.RX_FIFO_NOT_EMPTY CAN.RX_FIFO_OVERRUN CAN.FILTER_LIST CAN.FILTER_RANGE CAN.FILTER_MASK

class RMT – Remote Controller

The RMT (Remote Control) module is primarily designed to send and receive infrared remote control signals that use on-off-keying of a carrier frequency, but due to its design it can be used to generate various types of signals.

Quick Usage Example: sending

```
import machine

# create a RMT object for transmission
rmt = machine.RMT(channel=3, gpio="P20", tx_idle_level=0)
# create series of bits to send
data = (1,0,1,0,1,0,1,0,1)
# define duration of the bits, time unit depends on the selected RMT channel
duration = 10000
# send the signal
rmt.send_pulses(duration, data)
```

Quick Usage Example: receiving

```
import machine
# create a RMT object
rmt = machine.RMT(channel=3)
# Configure RTM for receiving
rmt.init(gpio="P20", rx_idle_threshold=12000)
# wait for any number of pulses until one longer than rx_idle_threshold
data = rmt.recv_pulses()
```

Constructors

class machine.RMT(channel,...)
Construct an RMT object on the given channel. `channel` can be 2-7. With no additional parameters, the RMT object is created but not initialised. If extra arguments are given, the RMT is initialised for transmission or reception. See `init` for parameters of initialisation. The resolution which a pulse can be sent/received depends on the selected channel:

Channel	Resolution	Maximum Pulse Width
0	Used by on-board LED	
1	Used by `pycom.pulses_get()`	
2	100nS	3.2768 ms
3	100nS	3.2768 ms
4	1000nS	32.768 ms
5	1000nS	32.768 ms
6	3125nS	102.4 ms
7	3125nS	102.4 ms

Methods

rmt.init(gpio, rx_idle_threshold, rx_filter_threshold, tx_idle_level, tx_carrier)
Initialise the RMT peripheral with the given parameters:

- `gpio` is the GPIO Pin to use.
- `rx_idle_threshold` is the maximum duration of a valid pulse. The represented time unit (resolution) depends on the selected channel, value can be 0-65535.
- `rx_filter_threshold` is the minimum duration of a valid pulse. The represented time unit (resolution) depends on the selected channel, value can be 0-31.
- `tx_idle_level` is the output signal's level after the transmission is finished, can be RMT.HIGH or RMT.LOW.
- `tx_carrier` is the modulation of the pulses to send.

Either `rx_idle_threshold` or `tx_idle_level` must be defined, both cannot be given at the same time because a channel can be configured in RX or TX mode only. `rx_filter_threshold` is not mandatory parameter. If not given then all pulses are accepted with duration less than `rx_idle_threshold`. `tx_carrier` is not mandatory parameters. If not given no modulation is used on the sent pulses.

The `tx_carrier` parameter is a tuple with the following structure:

- `carrier_freq_hz` is the carrier's frequency in Hz.
- `carrier_duty_percent` is the duty percent of the carrier's signal, can be 0%-100%.
- `carrier_level` is the level of the pulse to modulate, can be RMT.HIGH or RMT.LOW.

rmt.deinit()
Deinitialise the RMT object.

> If an RMT object needs to be reconfigured from RX/TX to TX/RX, then either first `deinit()` must be called or the `init()` again with the desired configuration.

rmt.pulses_get(pulses, timeout)
Reads in pulses from the GPIO pin.

- `pulses` if not specified, this function will keep reading pulses until the `rx_idle_threshold` is exceeded. If it is specified this function will return the exactly that number of pulses, ignoring anything shorter than `rx_filter_threshold` or longer than `rx_idle_threshold`.
- `timeout` is specified, this function will return if the first pulse does not occur within `timeout` microseconds. If not specified, it will wait indefinitely.

Return value: Tuple of items with the following structure: (level, duration):

- `level` represents the level of the received bit/pulse, can be 0 or 1.
- `duration` represents the duration of the received pulse, the time unit (resolution) depends on the selected channel.

> Maximum of 128 pulses can be received in a row without receiving "idle" signal. If the incoming pulse sequence contains more than 128 pulses the rest is dropped and the receiver waits for another sequence of pulses. The `pulses_get` function can be called to receive more than 128 pulses, however the above mentioned limitation should be kept in mind when evaluating the received data.

rmt.pulses_send(duration, data, start_level)
Generates pulses as defined by the parameters below

- `duration` represents the duration of the pulses to be sent, the time unit (resolution) depends on the selected channel.
- `data` Tuple that represents the sequence of pulses to be sent, must be composed of 0 or 1 elements.
- `start_level` defines the state (HIGH/LOW) of the first pulse given by `duration` if `data` is not given.

`data` must be a tuple and `duration` can be a tuple or a single number, with `data` being optional. In the case that only `duration` is provided, it must be a tuple and you must also provide `start_level` which will dictate the level of the first duration, the signal level then toggles between each duration value. If `data` is provided and `duration` is a single number, each pulse in `data` will have have an equal length as set by `duration`. If `data` and `duration` are provided as tuples, they must be of the same number of elements, with each pulse lasting its matching duration.

Constants

RMT.LOW RMT.HIGH
Defines the level of the pulse.

module network

This module provides access to network drivers and routing configuration. Network drivers for specific hardware are available within this module and are used to configure specific hardware network interfaces.

class WLAN

This class provides a driver for the WiFi network processor in the module. Example usage:

```
import network
import time
# setup as a station
wlan = network.WLAN(mode=network.WLAN.STA)
wlan.connect('your-ssid', auth=(network.WLAN.WPA2, 'your-key'))
while not wlan.isconnected():
    time.sleep_ms(50)
print(wlan.ifconfig())

# now use socket as usual
```

Quick Usage Example

```
import machine
from network import WLAN

# configure the WLAN subsystem in station mode (the default is AP)
wlan = WLAN(mode=WLAN.STA)
# go for fixed IP settings (IP, Subnet, Gateway, DNS)
wlan.ifconfig(config=('192.168.0.107', '255.255.255.0', '192.168.0.1', '192.168.0.1'))
wlan.scan()     # scan for available networks
wlan.connect(ssid='mynetwork', auth=(WLAN.WPA2, 'my_network_key'))
while not wlan.isconnected():
    pass
print(wlan.ifconfig())
```

Constructors

class network.WLAN(id=0, ...)

Create a WLAN object, and optionally configure it. See init for params of configuration.

> The WLAN constructor is special in the sense that if no arguments besides the `id` are given, it will return the already existing WLAN instance without re-configuring it. This is because WLAN is a system feature of the WiPy. If the already existing instance is not initialised it will do the same as the other constructors an will initialise it with default values.

Methods

wlan.init(mode, * , ssid=None, auth=None, channel=1, antenna=None, power_save=False, hidden=False)

Set or get the WiFi network processor configuration.

Arguments are:

- `mode` can be either WLAN.STA, WLAN.AP or WLAN.STA_AP.
- `ssid` is a string with the SSID name. Only needed when mode is WLAN.AP.
- `auth` is a tuple with (sec, key). Security can be `None`, WLAN.WEP, WLAN.WPA or WLAN.WPA2. The key is a string with the network password. If `sec` is WLAN.WEP the key must be a string representing hexadecimal values (e.g. `ABC1DE45BF`). Only needed when mode is WLAN.AP.
- `channel` a number in the range 1-11. Only needed when mode is WLAN.AP.
- `antenna` selects between the internal and the external antenna. Can be either WLAN.INT_ANT, WLAN.EXT_ANT. With our development boards it defaults to using the internal antenna, but in the case of an OEM module, the antenna pin (`P12`) is not used, so it's free to be used for other things.
- `power_save` enables or disables power save functions in STA mode.
- `hidden` only valid in WLAN.AP mode to create an access point with a hidden SSID when set to `True`.

For example, you can do:

```
# create and configure as an access point
wlan.init(mode=WLAN.AP, ssid='wipy-wlan', auth=(WLAN.WPA2,'www.wipy.io'), channel=7, a
ntenna=WLAN.INT_ANT)
```

or:

```
# configure as an station
wlan.init(mode=WLAN.STA)
```

wlan.deinit()

Disables the WiFi radio.

wlan.connect(ssid, * , auth=None, bssid=None, timeout=None, ca_certs=None, keyfile=None, certfile=None, identity=None)

Connect to a wifi access point using the given SSID, and other security parameters.

- `auth` is a tuple with `(sec, key)`. Security can be `None`, WLAN.WEP, WLAN.WPA, WLAN.WPA2 or WLAN.WPA2_ENT. The key is a string with the network password. If `sec` is WLAN.WEP the key must be a string representing hexadecimal values (e.g. `ABC1DE45BF`). If `sec` is WLAN.WPA2_ENT then the `auth` tuple can have either 3 elements: `(sec, username, password)`, or just 1: `(sec,)`. When passing the 3 element tuple, the `keyfile` and `certfile` arguments must not be given.
- `bssid` is the MAC address of the AP to connect to. Useful when there are several APs with the same SSID.
- `timeout` is the maximum time in milliseconds to wait for the connection to succeed.
- `ca_certs` is the path to the CA certificate. This argument is not mandatory. `keyfile` is the path to the client key. Only used if `username` and `password` are not part of the `auth` tuple.
- `certfile` is the path to the client certificate. Only used if `username` and `password` are not part of the `auth` tuple.
- `identity` is only used in case of WLAN.WPA2_ENT security.

wlan.scan()

Performs a network scan and returns a list of named tuples with `(ssid, bssid, sec, channel, rssi)`. Note that channel is always `None` since this info is not provided by the WiPy.

wlan.disconnect()

Disconnect from the WiFi access point.

wlan.isconnected()

In case of STA mode, returns `True` if connected to a WiFi access point and has a valid IP address. In AP mode returns `True` when a station is connected, `False` otherwise.

wlan.ifconfig(id=0, config=['dhcp' or configtuple])

When `id` is 0, the configuration will be get/set on the Station interface. When `id` is 1 the configuration will be done for the AP interface.

With no parameters given returns a 4-tuple of `(ip, subnet_mask, gateway, DNS_server)`.

If `dhcp` is passed as a parameter then the DHCP client is enabled and the IP params are negotiated with the AP.

If the 4-tuple config is given then a static IP is configured. For instance:

```
wlan.ifconfig(config=('192.168.0.4', '255.255.255.0', '192.168.0.1', '8.8.8.8'))
```

wlan.mode([mode])

Get or set the WLAN mode.

wlan.ssid([ssid])

Get or set the SSID when in AP mode.

wlan.auth([auth])

Get or set the authentication type when in AP mode.

wlan.channel([channel])

Get or set the channel (only applicable in AP mode).

wlan.antenna([antenna])

Get or set the antenna type (external or internal).

wlan.mac()

Get a 6-byte long `bytes` object with the WiFI MAC address.

Constants

WLAN.STA WLAN.AP WLAN.STA_AP
WLAN mode

WLAN.WEP WLAN.WPA WLAN.WPA2 WLAN.WPA2_ENT
WLAN network security

WLAN.INT_ANT WLAN.EXT_ANT
Antenna type

class Server

The `Server` class controls the behaviour and the configuration of the FTP and telnet services running on the Pycom device. Any changes performed using this class' methods will affect both.

Example:

```
import network
server = network.Server()
server.deinit() # disable the server
# enable the server again with new settings
server.init(login=('user', 'password'), timeout=600)
```

Quick Usage Example

```
from network import Server

# init with new user, password and seconds timeout
server = Server(login=('user', 'password'), timeout=60)
server.timeout(300) # change the timeout
server.timeout() # get the timeout
server.isrunning() # check whether the server is running or not
```

Constructors

class network.Server(id, ...)

Create a server instance, see `init` for parameters of initialisation.

Methods

server.init(* , login=('micro', 'python'), timeout=300)

Init (and effectively start the server). Optionally a new `user` , `password` and `timeout` (in seconds) can be passed.

server.deinit()

Stop the server.

server.timeout([timeout_in_seconds])

Get or set the server timeout.

server.isrunning()

Returns `True` if the server is running (connected or accepting connections), `False` otherwise.

class Bluetooth

This class provides a driver for the Bluetooth radio in the module. Currently, only basic BLE functionality is available.

Quick Usage Example

```
from network import Bluetooth
import time
bt = Bluetooth()
bt.start_scan(-1)

while True:
  adv = bt.get_adv()
  if adv and bt.resolve_adv_data(adv.data, Bluetooth.ADV_NAME_CMPL) == 'Heart Rate':
      try:
          conn = bt.connect(adv.mac)
          services = conn.services()
          for service in services:
              time.sleep(0.050)
              if type(service.uuid()) == bytes:
                  print('Reading chars from service = {}'.format(service.uuid()))
              else:
                  print('Reading chars from service = %x' % service.uuid())
              chars = service.characteristics()
              for char in chars:
                  if (char.properties() & Bluetooth.PROP_READ):
                      print('char {} value = {}'.format(char.uuid(), char.read()))
          conn.disconnect()
          break
      except:
          print("Error while connecting or reading from the BLE device")
          break
  else:
      time.sleep(0.050)
```

Bluetooth Low Energy (BLE)

Bluetooth low energy (BLE) is a subset of classic Bluetooth, designed for easy connecting and communicating between devices (in particular mobile platforms). BLE uses a methodology known as Generic Access Profile (GAP) to control connections and advertising.

GAP allows for devices to take various roles but generic flow works with devices that are either a Server (low power, resource constrained, sending small payloads of data) or a Client device (commonly a mobile device, PC or Pycom Device with large resources and processing power). Pycom devices can act as both a Client and a Server.

Constructors

class network.Bluetooth(id=0, ...)

Create a Bluetooth object, and optionally configure it. See init for params of configuration.

Example:

```
from network import Bluetooth
bluetooth = Bluetooth()
```

Methods

bluetooth.init(id=0, mode=Bluetooth.BLE, antenna=None)

- `id` Only one Bluetooth peripheral available so must always be 0
- `mode` currently the only supported mode is `Bluetooth.BLE`
- `antenna` selects between the internal and the external antenna. Can be either Bluetooth.INT_ANT, Bluetooth.EXT_ANT. With our development boards it defaults to using the internal antenna, but in the case of an OEM module, the antenna pin (`P12`) is not used, so it's free to be used for other things.

Initialises and enables the Bluetooth radio in BLE mode.

bluetooth.deinit()

Disables the Bluetooth radio.

bluetooth.start_scan(timeout)

Starts performing a scan listening for BLE devices sending advertisements. This function always returns immediately, the scanning will be performed on the background. The return value is `None` . After starting the scan the function get_adv() can be used to retrieve the advertisements messages from the FIFO. The internal FIFO has space to cache 16 advertisements.

The arguments are:

- `timeout` specifies the amount of time in seconds to scan for advertisements, cannot be zero. If timeout is > 0, then the BLE radio will listen for advertisements until the specified value in seconds elapses. If timeout < 0, then there's no timeout at all, and stop_scan() needs to be called to cancel the scanning process.

Examples:

```
bluetooth.start_scan(10)        # starts scanning and stop after 10 seconds
bluetooth.start_scan(-1)        # starts scanning indefinitely until bluetooth.stop_sc
an() is called
```

bluetooth.stop_scan()

Stops an ongoing scanning process. Returns `None` .

bluetooth.isscanning()

Returns `True` if a Bluetooth scan is in progress. `False` otherwise.

bluetooth.get_adv()

Gets an named tuple with the advertisement data received during the scanning. The tuple has the following structure: `(mac, addr_type, adv_type, rssi, data)`

- `mac` is the 6-byte ling mac address of the device that sent the advertisement.
- `addr_type` is the address type. See the constants section below for more details.
- `adv_type` is the advertisement type received. See the constants section below fro more details.
- `rssi` is signed integer with the signal strength of the advertisement.
- `data` contains the complete 31 bytes of the advertisement message. In order to parse the data and get the specific types, the method resolve_adv_data() can be used.

Example for getting `mac` address of an advertiser:

```
import ubinascii

bluetooth = Bluetooth()
bluetooth.start_scan(20) # scan for 20 seconds

adv = bluetooth.get_adv() #
ubinascii.hexlify(adv.mac) # convert hexadecimal to ascii
```

bluetooth.get_advertisements()

Same as the `get_adv()` method, but this one returns a list with all the advertisements received.

bluetooth.resolve_adv_data(data, data_type)

Parses the advertisement data and returns the requested `data_type` if present. If the data type is not present, the function returns `None`.

Arguments:

- `data` is the bytes object with the complete advertisement data.
- `data_type` is the data type to resolve from from the advertisement data. See constants section below for details.

Example:

```
import ubinascii
from network import Bluetooth
bluetooth = Bluetooth()

bluetooth.start_scan(20)
while bluetooth.isscanning():
    adv = bluetooth.get_adv()
    if adv:
        # try to get the complete name
        print(bluetooth.resolve_adv_data(adv.data, Bluetooth.ADV_NAME_CMPL))

        mfg_data = bluetooth.resolve_adv_data(adv.data, Bluetooth.ADV_MANUFACTURER_DAT
A)

        if mfg_data:
            # try to get the manufacturer data (Apple's iBeacon data is sent here)
            print(ubinascii.hexlify(mfg_data))
```

bluetooth.connect(mac_addr)

Opens a BLE connection with the device specified by the `mac_addr` argument. This function blocks until the connection succeeds or fails. If the connections succeeds it returns a object of type `GATTCConnection`.

```
bluetooth.connect('112233eeddff') # mac address is accepted as a string
```

bluetooth.callback(trigger=None, handler=None, arg=None)

Creates a callback that will be executed when any of the triggers occurs. The arguments are:

- `trigger` can be either Bluetooth.NEW_ADV_EVENT, Bluetooth.CLIENT_CONNECTED or Bluetooth.CLIENT_DISCONNECTED
- `handler` is the function that will be executed when the callback is triggered.

- `arg` is the argument that gets passed to the callback. If nothing is given the bluetooth object itself is used.

An example of how this may be used can be seen in the bluetooth.events() method.

bluetooth.events()

Returns a value with bit flags identifying the events that have occurred since the last call. Calling this function clears the events.

Example of usage:

```
from network import Bluetooth

bluetooth = Bluetooth()
bluetooth.set_advertisement(name='LoPy', service_uuid=b'1234567890123456')

def conn_cb (bt_o):
    events = bt_o.events()   # this method returns the flags and clears the internal r
egistry
    if events & Bluetooth.CLIENT_CONNECTED:
        print("Client connected")
    elif events & Bluetooth.CLIENT_DISCONNECTED:
        print("Client disconnected")

bluetooth.callback(trigger=Bluetooth.CLIENT_CONNECTED | Bluetooth.CLIENT_DISCONNECTED,
 handler=conn_cb)

bluetooth.advertise(True)
```

bluetooth.set_advertisement(* , name=None, manufacturer_data=None, service_data=None, service_uuid=None)

Configure the data to be sent while advertising. If left with the default of `None` the data won't be part of the advertisement message.

The arguments are:

- `name` is the string name to be shown on advertisements.
- `manufacturer_data` manufacturer data to be advertised (hint: use it for iBeacons).
- `service_data` service data to be advertised.
- `service_uuid` uuid of the service to be advertised.

Example:

```
bluetooth.set_advertisement(name="advert", manufacturer_data="lopy_v1")
```

bluetooth.advertise([Enable])

Start or stop sending advertisements. The set_advertisement() method must have been called prior to this one.

bluetooth.service(uuid, * , isprimary=True, nbr_chars=1, start=True)

Create a new service on the internal GATT server. Returns a object of type `BluetoothServerService`.

The arguments are:

- `uuid` is the UUID of the service. Can take an integer or a 16 byte long string or bytes object.
- `isprimary` selects if the service is a primary one. Takes a `bool` value.
- `nbr_chars` specifies the number of characteristics that the service will contain.
- `start` if `True` the service is started immediately.

```
bluetooth.service('abc123')
```

bluetooth.disconnect_client()

Closes the BLE connection with the client.

Constants

Bluetooth mode

Bluetooth.BLE

Advertisement type

Bluetooth.CONN_ADV Bluetooth.CONN_DIR_ADV Bluetooth.DISC_ADV Bluetooth.NON_CONN_ADV Bluetooth.SCAN_RSP

Address type

Bluetooth.PUBLIC_ADDR Bluetooth.RANDOM_ADDR Bluetooth.PUBLIC_RPA_ADDR Bluetooth.RANDOM_RPA_ADDR

Advertisement data type

Bluetooth.ADV_FLAG Bluetooth.ADV_16SRV_PART Bluetooth.ADV_T16SRV_CMPL Bluetooth.ADV_32SRV_PART Bluetooth.ADV_32SRV_CMPL Bluetooth.ADV_128SRV_PART Bluetooth.ADV_128SRV_CMPL

Bluetooth.ADV_NAME_SHORT Bluetooth.ADV_NAME_CMPL Bluetooth.ADV_TX_PWR Bluetooth.ADV_DEV_CLASS Bluetooth.ADV_SERVICE_DATA Bluetooth.ADV_APPEARANCE Bluetooth.ADV_ADV_INT Bluetooth.ADV_32SERVICE_DATA Bluetooth.ADV_128SERVICE_DATA Bluetooth.ADV_MANUFACTURER_DATA

Characteristic properties (bit values that can be combined)

Bluetooth.PROP_BROADCAST Bluetooth.PROP_READ Bluetooth.PROP_WRITE_NR Bluetooth.PROP_WRITE Bluetooth.PROP_NOTIFY Bluetooth.PROP_INDICATE Bluetooth.PROP_AUTH Bluetooth.PROP_EXT_PROP

Characteristic callback events

Bluetooth.CHAR_READ_EVENT Bluetooth.CHAR_WRITE_EVENT Bluetooth.NEW_ADV_EVENT Bluetooth.CLIENT_CONNECTED Bluetooth.CLIENT_DISCONNECTED Bluetooth.CHAR_NOTIFY_EVENT

Antenna type

Bluetooth.INT_ANT Bluetooth.EXT_ANT

Generic Attribute

GATT stands for the Generic Attribute Profile and it defines the way that two Bluetooth Low Energy devices communicate between each other using concepts called Services and Characteristics. GATT uses a data protocol known as the Attribute Protocol (ATT), which is used to store/manage Services, Characteristics and related data in a lookup table.

GATT comes into use once a connection is established between two devices, meaning that the device will have already gone through the advertising process managed by GAP. It's important to remember that this connection is exclusive; i.e. that only one client is connected to one server at a time. This means that the client will stop advertising once a connection has been made. This remains the case, until the connection is broken or disconnected.

The GATT Server, which holds the ATT lookup data and service and characteristic definitions, and the GATT Client (the phone/tablet), which sends requests to this server.

class GATTCConnection

The GATT Client is the device that requests data from the server, otherwise known as the master device (commonly this might be a phone/tablet/PC). All transactions are initiated by the master, which receives a response from the slave.

connection.disconnect()

Closes the BLE connection. Returns `None`.

connection.isconnected()

Returns `True` if the connection is still open. `False` otherwise.

Example:

```
from network import Bluetooth
import ubinascii
bluetooth = Bluetooth()

# scan until we can connect to any BLE device around
bluetooth.start_scan(-1)
adv = None
while True:
    adv = bluetooth.get_adv()
    if adv:
        try:
            bluetooth.connect(adv.mac)
        except:
            # start scanning again
            bluetooth.start_scan(-1)
            continue
        break
print("Connected to device with addr = {}".format(ubinascii.hexlify(adv.mac)))
```

connection.services()

Performs a service search on the connected BLE peripheral (server) a returns a list containing objects of the class GATTCService if the search succeeds.

Example:

```
# assuming that a BLE connection is already open
services = connection.services()
print(services)
for service in services:
    print(service.uuid())
```

class GATTCService

Services are used to categorise data up into specific chunks of data known as characteristics. A service may have multiple characteristics, and each service has a unique numeric ID called a UUID.

The following class allows control over Client services.

service.isprimary()

Returns `True` if the service is a primary one. `False` otherwise.

service.uuid()

Returns the UUID of the service. In the case of 16-bit or 32-bit long UUIDs, the value returned is an integer, but for 128-bit long UUIDs the value returned is a bytes object.

service.instance()

Returns the instance ID of the service.

service.characteristics()

Performs a get characteristics request on the connected BLE peripheral a returns a list containing objects of the class GATTCCharacteristic if the request succeeds.

class GATTCCharacteristic

The smallest concept in GATT is the Characteristic, which encapsulates a single data point (though it may contain an array of related data, such as X/Y/Z values from a 3-axis accelerometer, longitude and latitude from a GPS, etc.).

The following class allows you to manage characteristics from a Client.

characteristic.uuid()

Returns the UUID of the service. In the case of 16-bit or 32-bit long UUIDs, the value returned is an integer, but for 128-bit long UUIDs the value returned is a bytes object.

characteristic.instance()

Returns the instance ID of the service.

characteristic.properties()

Returns an integer indicating the properties of the characteristic. Properties are represented by bit values that can be OR-ed together. See the constants section for more details.

characteristic.read()

Read the value of the characteristic, sending a request to the GATT server. Returns a bytes object representing the characteristic value.

characteristic.value()

Returns the locally stored value of the characteristic without sending a read request to the GATT server. If the characteristic value hasn't been read from the GATT server yet, the value returned will be 0.

characteristic.write(value)

Writes the given value on the characteristic. For now it only accepts bytes object representing the value to be written.

```
characteristic.write(b'x0f')
```

characteristic.callback(trigger=None, handler=None, arg=None)

This method allows to register for notifications on the characteristic.

- `trigger` can must be Bluetooth.CHAR_NOTIFY_EVENT.
- `handler` is the function that will be executed when the callback is triggered.
- `arg` is the argument that gets passed to the callback. If nothing is given, the characteristic object that owns the callback will be used.

class GATTSService

The GATT Server allows the device to act as a peripheral and hold its own ATT lookup data, server & characteristic definitions. In this mode, the device acts as a slave and a master must initiate a request.

Services are used to categorise data up into specific chunks of data known as characteristics. A service may have multiple characteristics, and each service has a unique numeric ID called a UUID.

The following class allows control over Server services.

service.start()

Starts the service if not already started.

service.stop()

Stops the service if previously started.

service.characteristic(uuid, * , permissions, properties, value)

Creates a new characteristic on the service. Returns an object of the class GATTSCharacteristic. The arguments are:

- `uuid` is the UUID of the service. Can take an integer or a 16 byte long string or bytes object.
- `permissions` configures the permissions of the characteristic. Takes an integer with a combination of the flags.
- `properties` sets the properties. Takes an integer with an OR-ed combination of the flags.
- `value` sets the initial value. Can take an integer, a string or a bytes object.

```
service.characteristic('temp', value=25)
```

class GATTSCharacteristic

The smallest concept in GATT is the Characteristic, which encapsulates a single data point (though it may contain an array of related data, such as X/Y/Z values from a 3-axis accelerometer, longitude and latitude from a GPS, etc.).

The following class allows you to manage Server characteristics.

characteristic.value([value])

Gets or sets the value of the characteristic. Can take an integer, a string or a bytes object.

```
characteristic.value(123) # set characteristic value to an integer with the value 123
characteristic.value() # get characteristic value
```

characteristic.callback(trigger=None, handler=None, arg=None)

Creates a callback that will be executed when any of the triggers occurs. The arguments are:

- `trigger` can be either Bluetooth.CHAR_READ_EVENT or Bluetooth.CHAR_WRITE_EVENT.
- `handler` is the function that will be executed when the callback is triggered.
- `arg` is the argument that gets passed to the callback. If nothing is given, the characteristic object that owns the callback will be used.

An example of how this could be implemented can be seen in the characteristic.events() section.

characteristic.events()

Returns a value with bit flags identifying the events that have occurred since the last call. Calling this function clears the events.

An example of advertising and creating services on the device:

```
from network import Bluetooth

bluetooth = Bluetooth()
bluetooth.set_advertisement(name='LoPy', service_uuid=b'1234567890123456')

def conn_cb (bt_o):
    events = bt_o.events()
    if  events & Bluetooth.CLIENT_CONNECTED:
        print("Client connected")
    elif events & Bluetooth.CLIENT_DISCONNECTED:
        print("Client disconnected")

bluetooth.callback(trigger=Bluetooth.CLIENT_CONNECTED | Bluetooth.CLIENT_DISCONNECTED,
 handler=conn_cb)

bluetooth.advertise(True)

srv1 = bluetooth.service(uuid=b'1234567890123456', isprimary=True)

chr1 = srv1.characteristic(uuid=b'ab34567890123456', value=5)

char1_read_counter = 0
def char1_cb_handler(chr):
    global char1_read_counter
    char1_read_counter += 1

    events = chr.events()
    if  events & Bluetooth.CHAR_WRITE_EVENT:
        print("Write request with value = {}".format(chr.value()))
    else:
        if char1_read_counter < 3:
            print('Read request on char 1')
        else:
            return 'ABC DEF'

char1_cb = chr1.callback(trigger=Bluetooth.CHAR_WRITE_EVENT | Bluetooth.CHAR_READ_EVEN
T, handler=char1_cb_handler)

srv2 = bluetooth.service(uuid=1234, isprimary=True)

chr2 = srv2.characteristic(uuid=4567, value=0x1234)
char2_read_counter = 0xF0
def char2_cb_handler(chr):
    global char2_read_counter
    char2_read_counter += 1
    if char2_read_counter > 0xF1:
        return char2_read_counter

char2_cb = chr2.callback(trigger=Bluetooth.CHAR_READ_EVENT, handler=char2_cb_handler)
```

class LoRa

This class provides a LoRaWAN 1.0.2 compliant driver for the LoRa network processor in the LoPy and FiPy. Below is an example demonstrating LoRaWAN Activation by Personalisation usage:

```
from network import LoRa
import socket
import ubinascii
import struct

# Initialise LoRa in LORAWAN mode.
# Please pick the region that matches where you are using the device:
# Asia = LoRa.AS923
# Australia = LoRa.AU915
# Europe = LoRa.EU868
# United States = LoRa.US915
lora = LoRa(mode=LoRa.LORAWAN, region=LoRa.EU868)

# create an ABP authentication params
dev_addr = struct.unpack(">l", binascii.unhexlify('00000005'))[0]
nwk_swkey = ubinascii.unhexlify('2B7E151628AED2A6ABF7158809CF4F3C')
app_swkey = ubinascii.unhexlify('2B7E151628AED2A6ABF7158809CF4F3C')

# join a network using ABP (Activation By Personalisation)
lora.join(activation=LoRa.ABP, auth=(dev_addr, nwk_swkey, app_swkey))

# create a LoRa socket
s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)

# set the LoRaWAN data rate
s.setsockopt(socket.SOL_LORA, socket.SO_DR, 5)

# make the socket non-blocking
s.setblocking(False)

# send some data
s.send(bytes([0x01, 0x02, 0x03]))

# get any data received...
data = s.recv(64)
print(data)
```

Please ensure that there is an antenna connected to your device before sending/receiving LoRa messages as improper use (e.g. without an antenna), may damage the device.

Additional Examples

For various other complete LoRa examples, check here for additional examples.

Constructors

***class* network.LoRa(id=0, ...)**

Create and configure a LoRa object. See init for params of configuration.

```
lora = LoRa(mode=LoRa.LORAWAN, region=LoRa.EU868)
```

Methods

lora.init(mode, * ,region=LoRa.EU868, frequency=868000000, tx_power=14, bandwidth=LoRa.BW_125KHZ, sf=7, preamble=8, coding_rate=LoRa.CODING_4_5, power_mode=LoRa.ALWAYS_ON, tx_iq=False, rx_iq=False, adr=False, public=True, tx_retries=1, device_class=LoRa.CLASS_A)

This method is used to set the LoRa subsystem configuration and to specific raw LoRa or LoRaWAN.

The arguments are:

- `mode` can be either LoRa.LORA or LoRa.LORAWAN.
- `region` can take the following values: LoRa.AS923, LoRa.AU915, LoRa.EU868 or LoRa.US915. If not provided this will default to `LoRaEU868` . If they are not specified, this will also set appropriate defaults for `frequency` and `tx_power` .
- `frequency` accepts values between 863000000 and 870000000 in the 868 band, or between 902000000 and 928000000 in the 915 band.
- `tx_power` is the transmit power in dBm. It accepts between 2 and 14 for the 868 band, and between 5 and 20 in the 915 band.
- `bandwidth` is the channel bandwidth in KHz. In the 868 band the accepted values are LoRa.BW_125KHZ and LoRa.BW_250KHZ. In the 915 band the accepted values are LoRa.BW_125KHZ and LoRa.BW_500KHZ.
- `sf` sets the desired spreading factor. Accepts values between 7 and 12.
- `preamble` configures the number of pre-amble symbols. The default value is 8.
- `coding_rate` can take the following values: LoRa.CODING_4_5, LoRa.CODING_4_6, LoRa.CODING_4_7 or LoRa.CODING_4_8.
- `power_mode` can be either LoRa.ALWAYS_ON, LoRa.TX_ONLY or LoRa.SLEEP. In

ALWAYS_ON mode, the radio is always listening for incoming - packets whenever a transmission is not taking place. In TX_ONLY the radio goes to sleep as soon as the transmission completes. In SLEEP mode the radio is sent to sleep permanently and won't accept any commands until the power mode is changed.
- `tx_iq` enables TX IQ inversion.
- `rx_iq` enables RX IQ inversion.
- `adr` enables Adaptive Data Rate.
- `public` selects between the public and private sync word.
- `tx_retries` sets the number of TX retries in LoRa.LORAWAN mode.
- `device_class` sets the LoRaWAN device class. Can be either LoRa.CLASS_A or LoRa.CLASS_C.

> In LoRa.LORAWAN mode, only `adr` , `public` , `tx_retries` and `device_class` are used. All the other params will be ignored as they are handled by the LoRaWAN stack directly. On the other hand, in LoRa.LORA mode from those 4 arguments, only the public one is important in order to program the sync word. In LoRa.LORA mode `adr` , `tx_retries` and `device_class` are ignored since they are only relevant to the LoRaWAN stack.

For example, you can do:

```
# initialize in raw LoRa mode
lora.init(mode=LoRa.LORA, tx_power=14, sf=12)
```

or:

```
# initialize in LoRaWAN mode
lora.init(mode=LoRa.LORAWAN)
```

lora.join(activation, auth, * ,timeout=None, dr=None)

Join a LoRaWAN network. Internally the stack will automatically retry every 15 seconds until a Join Accept message is received.

The parameters are:

- `activation` : can be either LoRa.OTAA or LoRa.ABP.
- `auth` : is a tuple with the authentication data.
- `timeout` : is the maximum time in milliseconds to wait for the Join Accept message to be received. If no timeout (or zero) is given, the call returns immediately and the status of the join request can be checked with `lora.has_joined()` .

- `dr` : is an optional value to specify the initial data rate for the Join Request. Possible values are 0 to 5 for **EU868**, or 0 to 4 for **US915**.

In the case of LoRa.OTAA the authentication tuple is: `(dev_eui, app_eui, app_key)` where `dev_eui` is optional. If it is not provided the LoRa MAC will be used. Therefore, you can do OTAA in 2 different ways:

```
lora.join(activation=LoRa.OTAA, auth=(app_eui, app_key), timeout=0)  # the device MAC address is used as DEV_EUI
```

or

```
lora.join(activation=LoRa.OTAA, auth=(dev_eui, app_eui, app_key), timeout=0) # a custom DEV_EUI is specified
```

Example:

```
from network import LoRa
import socket
import time
import ubinascii

# Initialise LoRa in LORAWAN mode.
# Please pick the region that matches where you are using the device:
# Asia = LoRa.AS923
# Australia = LoRa.AU915
# Europe = LoRa.EU868
# United States = LoRa.US915
lora = LoRa(mode=LoRa.LORAWAN, region=LoRa.EU868)

# create an OTAA authentication parameters
app_eui = ubinascii.unhexlify('ADA4DAE3AC12676B')
app_key = ubinascii.unhexlify('11B0282A189B75B0B4D2D8C7FA38548B')

# join a network using OTAA (Over the Air Activation)
lora.join(activation=LoRa.OTAA, auth=(app_eui, app_key), timeout=0)

# wait until the module has joined the network
while not lora.has_joined():
    time.sleep(2.5)
    print('Not yet joined...')
```

In the case of LoRa.ABP the authentication tuple is: `(dev_addr, nwk_swkey, app_swkey)` . Example:

```
from network import LoRa
import socket
import ubinascii
import struct

# Initialise LoRa in LORAWAN mode.
# Please pick the region that matches where you are using the device:
# Asia = LoRa.AS923
# Australia = LoRa.AU915
# Europe = LoRa.EU868
# United States = LoRa.US915
lora = LoRa(mode=LoRa.LORAWAN, region=LoRa.EU868)

# create an ABP authentication params
dev_addr = struct.unpack(">l", ubinascii.unhexlify('00000005'))[0]
nwk_swkey = ubinascii.unhexlify('2B7E151628AED2A6ABF7158809CF4F3C')
app_swkey = ubinascii.unhexlify('2B7E151628AED2A6ABF7158809CF4F3C')

# join a network using ABP (Activation By Personalisation)
lora.join(activation=LoRa.ABP, auth=(dev_addr, nwk_swkey, app_swkey))
```

lora.bandwidth([bandwidth])

Get or set the bandwidth in raw LoRa mode (LoRa.LORA). Can be either LoRa.BW_125KHZ (0), LoRa.BW_250KHZ (1) or LoRa.BW_500KHZ (2):

```
# get raw LoRa Bandwidth
lora.bandwidth()

# set raw LoRa Bandwidth
lora.bandwidth(LoRa.BW_125KHZ)
```

lora.frequency([frequency])

Get or set the frequency in raw LoRa mode (LoRa.LORA). The allowed range is between 863000000 and 870000000 Hz for the 868 MHz band version or between 902000000 and 928000000 Hz for the 915 MHz band version.

```
# get raw LoRa Frequency
lora.frequency()

# set raw LoRa Frequency
lora.frequency(868000000)
```

lora.coding_rate([coding_rate])

Get or set the coding rate in raw LoRa mode (LoRa.LORA). The allowed values are: LoRa.CODING_4_5 (1), LoRa.CODING_4_6 (2), LoRa.CODING_4_7 (3) and LoRa.CODING_4_8 (4).

```
# get raw LoRa Coding Rate
lora.coding_rate()

# set raw LoRa Coding Rate
lora.coding_rate(LoRa.CODING_4_5)
```

lora.preamble([preamble])

Get or set the number of preamble symbols in raw LoRa mode (LoRa.LORA):

```
# get raw LoRa preamble symbols
lora.preamble()

# set raw LoRa preamble symbols
lora.preamble(LoRa.CODING_4_5)
```

lora.sf([sf])

Get or set the spreading factor value in raw LoRa mode (LoRa.LORA). The minimum value is 7 and the maximum is 12:

```
# get raw LoRa spread factor value
lora.sf()

# set raw LoRa spread factor value
lora.sf(7)
```

lora.power_mode([power_mode])

Get or set the power mode in raw LoRa mode (LoRa.LORA). The accepted values are: LoRa.ALWAYS_ON, LoRa.TX_ONLY and LoRa.SLEEP:

lora.stats()

Return a named tuple with useful information from the last received LoRa or LoRaWAN packet. The named tuple has the following form:

`(rx_timestamp, rssi, snr, sftx, sfrx, tx_trials, tx_power, tx_time_on_air, tx_counter, tx_frequency)`

Example:

```
lora.stats()
```

Where:

- `rx_timestamp` is an internal timestamp of the last received packet with microseconds precision.
- `rssi` holds the received signal strength in dBm.
- `snr` contains the signal to noise ratio id dB (as a single precision float).
- `sfrx` tells the data rate (in the case of LORAWAN mode) or the spreading factor (in the case of LORA mode) of the last packet received.
- `sftx` tells the data rate (in the case of LORAWAN mode) or the spreading factor (in the case of LORA mode) of the last packet transmitted.
- `tx_trials` is the number of tx attempts of the last transmitted packet (only relevant for LORAWAN confirmed packets).
- `tx_power` is the power of the last transmission (in dBm).
- `tx_time_on_air` is the time on air of the last transmitted packet (in ms).
- `tx_counter` is the number of packets transmitted.
- `tx_frequency` is the frequency used for the last transmission.

lora.has_joined()

Returns `True` if a LoRaWAN network has been joined. `False` otherwise.:

lora.add_channel(index, * , frequency, dr_min, dr_max)

Add a LoRaWAN channel on the specified `index`. If there's already a channel with that index it will be replaced with the new one.

The arguments are:

- `index` : Index of the channel to add. Accepts values between 0 and 15 for EU and between 0 and 71 for US.
- `frequency` : Centre frequency in Hz of the channel.
- `dr_min` : Minimum data rate of the channel (0-7).
- `dr_max` : Maximum data rate of the channel (0-7).

Examples:

```
lora.add_channel(index=0, frequency=868000000, dr_min=5, dr_max=6)
```

lora.remove_channel(index)

Removes the channel from the specified `index`. On the 868MHz band the channels 0 to 2 cannot be removed, they can only be replaced by other channels using the lora.add_channel method. A way to remove all channels except for one is to add the same channel, 3 times on indexes 0, 1 and 2. An example can be seen below:

```
lora.remove_channel()
```

On the 915MHz band there are no restrictions around this.

lora.mac()

Returns a byte object with the 8-Byte MAC address of the LoRa radio.

lora.callback(trigger, handler=None, arg=None)

Specify a callback handler for the LoRa radio. The `trigger` types are LoRa.RX_PACKET_EVENT, LoRa.TX_PACKET_EVENT and LoRa.TX_FAILED_EVENT

The LoRa.RX_PACKET_EVENT event is raised for every received packet. The LoRa.TX_PACKET_EVENT event is raised as soon as the packet transmission cycle ends, which includes the end of the receive windows (even if a downlink is received, the LoRa.TX_PACKET_EVENT will come last). In the case of non-confirmed transmissions, this will occur at the end of the receive windows, but, in the case of confirmed transmissions, this event will only be raised if the `ack` is received. If the `ack` is not received LoRa.TX_FAILED_EVENT will be raised after the number of `tx_retries` configured have been performed.

An example of how this callback functions can be seen the in method lora.events().

lora.ischannel_free(rssi_threshold)

This method is used to check for radio activity on the current LoRa channel, and if the `rssi` of the measured activity is lower than the `rssi_threshold` given, the return value will be `True`, otherwise `False`. Example:

```
lora.ischannel_free(-100)
```

lora.set_battery_level(level)

Set the battery level value that will be sent when the LoRaWAN MAC command that retrieves the battery level is received. This command is sent by the network and handled automatically by the LoRaWAN stack. The values should be according to the LoRaWAN specification:

- `0` means that the end-device is connected to an external power source.
- `1..254` specifies the battery level, 1 being at minimum and 254 being at maximum.
- `255` means that the end-device was not able to measure the battery level.

```
lora.set_battery_level(127) # 50% battery
```

lora.events()

This method returns a value with bits sets (if any) indicating the events that have triggered the callback. Please note that by calling this function the internal events registry is cleared automatically, therefore calling it immediately for a second time will most likely return a value of 0.

Example:

```
def lora_cb(lora):
    events = lora.events()
    if events & LoRa.RX_PACKET_EVENT:
        print('Lora packet received')
    if events & LoRa.TX_PACKET_EVENT:
        print('Lora packet sent')

lora.callback(trigger=(LoRa.RX_PACKET_EVENT | LoRa.TX_PACKET_EVENT), handler=lora_cb)
```

lora.nvram_save()

Save the LoRaWAN state (joined status, network keys, packet counters, etc) in non-volatile memory in order to be able to restore the state when coming out of deepsleep or a power cycle.

```
lora.nvram_save()
```

lora.nvram_restore()

Restore the LoRaWAN state (joined status, network keys, packet counters, etc) from non-volatile memory. State must have been previously stored with a call to `nvram_save` before entering deepsleep. This is useful to be able to send a LoRaWAN message immediately after coming out of deepsleep without having to join the network again. This can only be used if the current region matches the one saved.

```
lora.nvram_restore()
```

lora.nvram_erase()

Remove the LoRaWAN state (joined status, network keys, packet counters, etc) from non-volatile memory.

```
lora.nvram_erase()
```

Constants

LoRa.LORA LoRa.LORAWAN LoRa stack mode

LoRa.OTAA LoRa.ABP LoRaWAN join procedure

LoRa.ALWAYS_ON LoRa.TX_ONLY LoRa.SLEEP Raw LoRa power mode

LoRa.BW_125KHZ LoRa.BW_250KHZ LoRa.BW_500KHZ Raw LoRa bandwidth

LoRa.CODING_4_5 LoRa.CODING_4_6 LoRa.CODING_4_7 LoRa.CODING_4_8 Raw LoRa coding rate

LoRa.RX_PACKET_EVENT LoRa.TX_PACKET_EVENT LoRa.TX_FAILED_EVENT Callback trigger types (may be ORed)

LoRa.CLASS_A LoRa.CLASS_C LoRaWAN device class

LoRa.AS923 LoRa.AU915 LoRa.EU868 LoRa.US915 LoRaWAN regions

Working with LoRa and LoRaWAN Sockets

LoRa sockets are created in the following way:

```
import socket
s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
```

And they must be created after initialising the LoRa network card.

LoRa sockets support the following standard methods from the socket module:

socket.close()

Usage:

```
s.close()
```

socket.bind(port_number)

Usage:

```
s.bind(1)
```

The bind() method is only applicable when the radio is configured in LoRa.LORAWAN mode.

socket.send(bytes)

Usage:

```
s.send(bytes([1, 2, 3]))
```

or:

```
s.send('Hello')
```

socket.recv(bufsize)

Usage:

```
s.recv(128)
```

socket.recvfrom(bufsize)

This method is useful to know the destination port number of the message received. Returns a tuple of the form: `(data, port)`

Usage:

```
s.recvfrom(128)
```

socket.setsockopt(level, optname, value)

Set the value of the given socket option. The needed symbolic constants are defined in the socket module (`SO_*` etc.). In the case of LoRa the values are always integers. Examples:

```
# configuring the data rate
s.setsockopt(socket.SOL_LORA, socket.SO_DR, 5)

# selecting non-confirmed type of messages
s.setsockopt(socket.SOL_LORA, socket.SO_CONFIRMED, False)

# selecting confirmed type of messages
s.setsockopt(socket.SOL_LORA, socket.SO_CONFIRMED, True)
```

Socket options are only applicable when the LoRa radio is used in LoRa.LORAWAN mode. When using the radio in LoRa.LORA mode, use the class methods to change the spreading factor, bandwidth and coding rate to the desired values.

socket.settimeout(value)

Sets the socket timeout value in seconds. Accepts floating point values.

Usage:

```
s.settimeout(5.5)
```

socket.setblocking(flag)

Usage:

```
s.setblocking(True)
```

class Sigfox

Sigfox is a Low Power Wide Area Network protocol that enables remote devices to connect using ultra-narrow band, UNB technology. The protocol is bi-directional, messages can both be sent up to and down from the Sigfox servers.

When operating in `RCZ2` and `RCZ4` the module can only send messages on the default macro-channel (this is due to Sigfox network limitations). Therefore, the device needs to reset automatically to the default macro-channel after every 2 transmissions. However, due to FCC duty cycle limitations, there must a minimum of a 20s delay after resetting to the default macro-channel. Our API takes care of this, (and in real life applications you should not be in the need to send Sigfox messages that often), so it will wait for the necessary amount of time to make sure that the duty cycle restrictions are fulfilled.

This means that if you run a piece of test code like:

```
for i in range(1, 100):
  # send something
  s.send('Hello ' + str(i))
```

There will be a 20 second delay after every 2 packets.

This class provides a driver for the Sigfox network processor in the Sigfox enabled Pycom devices.

Quick Usage Example

```
from network import Sigfox
import socket

# init Sigfox for RCZ1 (Europe)
sigfox = Sigfox(mode=Sigfox.SIGFOX, rcz=Sigfox.RCZ1)

# create a Sigfox socket
s = socket.socket(socket.AF_SIGFOX, socket.SOCK_RAW)

# make the socket blocking
s.setblocking(True)

# configure it as uplink only
s.setsockopt(socket.SOL_SIGFOX, socket.SO_RX, False)

# send some bytes
s.send(bytes([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]))
```

Please ensure that there is an antenna connected to your device before sending/receiving Sigfox messages as in proper use (e.g. without an antenna), may damage the device.

Constructors

class network.Sigfox(id=0, ...)

Create and configure a Sigfox object. See init for params of configuration. Examples:

```
# configure radio for the Sigfox network, using RCZ1 (868 MHz)
sigfox = Sigfox(mode=Sigfox.SIGFOX, rcz=Sigfox.RCZ1)

# configure radio for FSK, device to device across 912 MHz
sigfox = Sigfox(mode=Sigfox.FSK, frequency=912000000)
```

Methods

sigfox.init(mode=Sigfox.SIGFOX, rcz=Sigfox.RCZ1, * , frequency=None)

Set the Sigfox radio configuration.

The arguments are:

- `mode` can be either Sigfox.SIGFOX or Sigfox.FSK. Sigfox.SIGFOX uses the Sigfox modulation and protocol while Sigfox.FSK allows to create point to point communication between 2 Devices using FSK modulation.

- `rcz` takes the following values: Sigfox.RCZ1, Sigfox.RCZ2, Sigfox.RCZ3, Sigfox.RCZ4. The `rcz` argument is only required if the mode is Sigfox.SIGFOX.
- `frequency` sets the frequency value in `FSK` mode. Can take values between 863 and 928 MHz.

> The SiPy comes in 2 different hardware flavours: a +14dBm Tx power version which can only work with `RCZ1` and `RCZ3` and a +22dBm version which works exclusively on `RCZ2` and `RCZ4`.

sigfox.mac()

Returns a byte object with the 8-Byte MAC address of the Sigfox radio.

sigfox.id()

Returns a byte object with the 4-Byte bytes object with the Sigfox ID.

sigfox.rssi()

Returns a signed integer with indicating the signal strength value of the last received packet.

sigfox.pac()

Returns a byte object with the 8-Byte bytes object with the Sigfox PAC.

> To return human-readable values you should import `ubinascii` and convert binary values to hexidecimal representation. For example:

```
print(ubinascii.hexlify(sigfox.mac()))
```

sigfox.frequencies()

Returns a tuple of the form: `(uplink_frequency_hz, downlink_frequency_hz)`

sigfox.public_key([public])

Sets or gets the public key flag. When called passing a `True` value the Sigfox public key will be used to encrypt the packets. Calling it without arguments returns the state of the flag.

```
# enable encrypted packets
sigfox.public_key(True)

# return state of public_key
sigfox.public_key()
```

Constants

sigfox.SIGFOX sigfox.FSK Sigfox radio mode. SIGFOX to specify usage of the Sigfox Public Network. `FSK` to specify device to device communication.

sigfox.RCZ1 sigfox.RCZ2 sigfox.RCZ3 sigfox.RCZ4 Sigfox zones.

- `RCZ1` to specify Europe, Oman & South Africa.
- `RCZ2` for the USA, Mexico & Brazil. RCZ3 for Japan.
- `RCZ4` for Australia, New Zealand, Singapore, Taiwan, Hong Kong, Colombia & Argentina.

Working with Sigfox Sockets

Sigfox sockets are created in the following way:

```
import socket
s = socket.socket(socket.AF_SIGFOX, socket.SOCK_RAW)
```

And they must be created after initialising the Sigfox network card.

Sigfox sockets support the following standard methods from the `socket` module:

socket.close()

Use it to close an existing socket.

socket.send(bytes)

In Sigfox mode the maximum data size is 12 bytes. In FSK the maximum is 64.

```
# send a Sigfox payload of bytes
s.send(bytes([1, 2, 3]))

# send a Sigfox payload containing a string
s.send('Hello')
```

socket.recv(bufsize)

This method can be used to receive a Sigfox downlink or FSK message.

```
# size of buffer should be passed for expected payload, e.g. 64 bytes
s.recv(64)
```

socket.setsockopt(level, optname, value)

Set the value of the given socket option. The needed symbolic constants are defined in the socket module (`SO_*` etc.). In the case of Sigfox the values are always an integer. Examples:

```
# wait for a downlink after sending the uplink packet
s.setsockopt(socket.SOL_SIGFOX, socket.SO_RX, True)

# make the socket uplink only
s.setsockopt(socket.SOL_SIGFOX, socket.SO_RX, False)

# use the socket to send a Sigfox Out Of Band message
s.setsockopt(socket.SOL_SIGFOX, socket.SO_OOB, True)

# disable Out-Of-Band to use the socket normally
s.setsockopt(socket.SOL_SIGFOX, socket.SO_OOB, False)

# select the bit value when sending bit only packets
s.setsockopt(socket.SOL_SIGFOX, socket.SO_BIT, False)
```

Sending a Sigfox packet with a single bit is achieved by sending an empty string, i.e.:

```

import socket
s = socket.socket(socket.AF_SIGFOX, socket.SOCK_RAW)

# send a 1 bit
s.setsockopt(socket.SOL_SIGFOX, socket.SO_BIT, True)
s.send('')
socket.settimeout(value)
# set timeout for the socket, e.g. 5 seconds
s.settimeout(5.0)
socket.setblocking(flag)
# specifies if socket should be blocking based upon Boolean flag.
s.setblocking(True)
```

If the socket is set to blocking, your code will be wait until the socket completes sending/receiving.

Sigfox Downlink

A Sigfox capable Pycom devices (SiPy) can both send and receive data from the Sigfox network. To receive data, a message must first be sent up to Sigfox, requesting a downlink message. This can be done by passing a `True` argument into the `setsockopt()` method.

```
s.setsockopt(socket.SOL_SIGFOX, socket.SO_RX, True)
```

An example of the downlink procedure can be seen below:

```
# init Sigfox for RCZ1 (Europe)
sigfox = Sigfox(mode=Sigfox.SIGFOX, rcz=Sigfox.RCZ1)

# create a Sigfox socket
s = socket.socket(socket.AF_SIGFOX, socket.SOCK_RAW)

# make the socket blocking
s.setblocking(True)

# configure it as DOWNLINK specified by 'True'
s.setsockopt(socket.SOL_SIGFOX, socket.SO_RX, True)

# send some bytes and request DOWNLINK
s.send(bytes([1, 2, 3]))

# await DOWNLINK message
s.recv(32)
```

Sigfox FSK (Device to Device)

To communicate between two Sigfox capable devices, it may be used in FSK mode. Two devices are required to be set to the same frequency, both using FSK.

Device 1:

```
sigfox = Sigfox(mode=Sigfox.FSK, frequency=868000000)

s = socket.socket(socket.AF_SIGFOX, socket.SOCK_RAW)
s.setblocking(True)

while True:
  s.send('Device-1')
  time.sleep(1)
  print(s.recv(64))
```

Device 2:

```
sigfox = Sigfox(mode=Sigfox.FSK, frequency=868000000)

s = socket.socket(socket.AF_SIGFOX, socket.SOCK_RAW)
s.setblocking(True)

while True:
  s.send('Device-2')
  time.sleep(1)
  print(s.recv(64))
```

Remember to use the correct frequency for your region (868 MHz for Europe, 912 MHz for USA, etc.)

class LTE

The LTE class provides access to the LTE-M/NB-IoT modem on the GPy and FiPy. LTE-M/NB-IoT are new categories of cellular protocols developed by the 3GPP and optimised for long battery life power and longer range. These are new protocols currently in the process of being deployed by mobile networks across the world.

The GPy and FiPy support both new LTE-M protocols:

- **Cat-M1**: also known as **LTE-M** defines a 1.4 MHz radio channel size and about 375 kbps of throughput. It is optimised for coverage and long battery life, outperforming 2G/GPRS, while being similar to previous LTE standards.
- **Cat-NB1** also known as **NB-IoT**, defines a 200 kHz radio channel size and around 60 kbps of uplink speed. It's optimised for ultra low throughput and specifically designed for IoT devices with a very long battery life. NB-IoT shares some features with LTE such as operating in licensed spectrum, but it's a very different protocol. It should be noted that NB-IoT has many restrictions as does not offer full IP connectivity and does not support mobility. When moving between cells, you will need to reconnect.

Please note: The GPy and FiPy only support the two protocols above and are not compatible with older LTE standards.

The Sequans modem used on Pycom's cellular enabled modules can only work in one of these modes at a time. In order to switch between the two protocols you need to flash a different firmware to the Sequans modem. Instructions for this can be found here.

AT Commands

The AT commands for the Sequans Monarch modem on the GPy/FiPy are available in a PDF file.

Constructors

***class* network.LTE(id=0, ...)**

Create and configure a LTE object. See init for params of configuration.

```
from network import LTE
lte = LTE()
```

Methods

lte.init(*, carrier=None)

This method is used to set up the LTE subsystem. After a `deinit()` this method can take several seconds to return waiting for the LTE modem to start-up. Optionally specify a carrier name. The available options are: `verizon, at&t, standard`. `standard` is generic for any carrier, and it's also the option used when no arguments are given.

lte.deinit()

Disables LTE modem completely. This reduces the power consumption to the minimum. Call this before entering deepsleep.

lte.attach(*, band=None)

Enable radio functionality and attach to the LTE Cat M1 network authorised by the inserted SIM card. Optionally specify the band to scan for networks. If no band (or `None`) is specified, all 6 bands will be scanned. The possible values for the band are: `3, 4, 12, 13, 20 and 28`.

lte.isattached()

Returns `True` if the cellular mode is attached to the network. `False` otherwise.

lte.dettach()

Detach the modem from the LTE Cat M1 and disable the radio functionality.

lte.connect(*, cid=1)

Start a data session and obtain and IP address. Optionally specify a CID (Connection ID) for the data session. The arguments are:

```
- `cid` is a Connection ID. This is carrier specific, for Verizon use `cid=3`. For oth
ers like Telstra it should be `cid=1`.
```

For instance, to attach and connect to Verizon:

```
import time
from network import LTE

lte = LTE(carrier="verizon")
lte.attach(band=13)

while not lte.isattached():
    time.sleep(0.5)
    print('Attaching...')

lte.connect(cid=3)
while not lte.isconnected():
    time.sleep(0.5)
    print('Connecting...')

# Now use sockets as usual...
```

lte.isconnected()

Returns `True` if there is an active LTE data session and IP address has been obtained. `False` otherwise.

lte.disconnect()

End the data session with the network.

lte.send_at_cmd(cmd)

Send an AT command directly to the modem. Returns the raw response from the modem as a string object. **IMPORTANT:** If a data session is active (i.e. the modem is *connected*), sending the AT commands requires to pause and then resume the data session. This is all done automatically, but makes the whole request take around 2.5 seconds.

Example:

```
lte.send_at_cmd('AT+CEREG?')    # check for network registration manually (sames as lt
e.isattached())
```

Optionally the response can be parsed for pretty printing:

```
def send_at_cmd_pretty(cmd):
    response = lte.send_at_cmd(cmd).split('\r\n')
    for line in response:
        print(line)

send_at_cmd_pretty('AT!="showphy"')     # get the PHY status
send_at_cmd_pretty('AT!="fsm"')         # get the System FSM
```

lte.imei()

Returns a string object with the IMEI number of the LTE modem.

lte.iccid()

Returns a string object with the ICCID number of the SIM card.

lte.reset()

Perform a hardware reset on the cellular modem. This function can take up to 5 seconds to return as it waits for the modem to shutdown and reboot.

class AES - Advanced Encryption Standard

AES (Advanced Encryption Standard) is a symmetric block cipher standardised by NIST. It has a fixed data block size of 16 bytes. Its keys can be 128, 192, or 256 bits long.

AES is implemented using the ESP32 hardware module.

Quick Usage Example

```
from crypto import AES
import crypto
key = b'notsuchsecretkey' # 128 bit (16 bytes) key
iv = crypto.getrandbits(128) # hardware generated random IV (never reuse it)

cipher = AES(key, AES.MODE_CFB, iv)
msg = iv + cipher.encrypt(b'Attack at dawn')

# ... after properly sent the encrypted message somewhere ...

cipher = AES(key, AES.MODE_CFB, msg[:16]) # on the decryption side
original = cipher.decrypt(msg[16:])
print(original)
```

Constructors

class ucrypto.AES(key, mode, IV, * , counter, segment_size)

Create an AES object that will let you encrypt and decrypt messages.

The arguments are:

- `key` (byte string) is the secret key to use. It must be 16 (AES-128), 24 (AES-192), or 32 (AES-256) bytes long.
- `mode` is the chaining mode to use for encryption and decryption. Default is AES.MODE_ECB.
- `IV` (byte string) initialisation vector. Should be 16 bytes long. It is ignored in modes AES.MODE_ECB and AES.MODE_CRT.
- `counter` (byte string) used only for AES.MODE_CTR. Should be 16 bytes long. Should not be reused.

- `segment_size` is the number of bits `plaintext` and `ciphertext` are segmented in. Is only used in AES.MODE_CFB. Supported values are AES.SEGMENT_8 and AES.SEGMENT_128.

Methods

ucrypto.encrypt()

Encrypt data with the key and the parameters set at initialisation.

ucrypto.decrypt()

Decrypt data with the key and the parameters set at initialisation.

Constants

AES.MODE_ECB
Electronic Code Book. Simplest encryption mode. It does not hide data patterns well (see this article for more info).

AES.MODE_CBC
Cipher-Block Chaining. An Initialisation Vector (IV) is required.

AES.MODE_CFB
Cipher feedback. `plaintext` and `ciphertext` are processed in segments of `segment_size` bits. Works a stream cipher.

AES.MODE_CTR
Counter mode. Each message block is associated to a counter which must be unique across all messages that get encrypted with the same key.

AES.SEGMENT_8 AES.SEGMENT_128
Length of the segment for AES.MODE_CFB.

> To avoid security issues, IV should always be a random number and should never be reused to encrypt two different messages. The same applies to the counter in CTR mode. You can use crypto.getrandbits() for this purpose.

pycom – Pycom Device Features

The `pycom` module contains functions to control specific features of the Pycom devices, such as the heartbeat RGB LED.

Quick Usage Example

```
import pycom

pycom.heartbeat(False)  # disable the heartbeat LED
pycom.heartbeat(True)   # enable the heartbeat LED
pycom.heartbeat()       # get the heartbeat state
pycom.rgbled(0xff00)    # make the LED light up in green color
```

Functions

pycom.heartbeat([enable])

Get or set the state (enabled or disabled) of the heartbeat LED. Accepts and returns boolean values (`True` or `False`).

pycom.heartbeat_on_boot([enable])

Allows you permanently disable or enable the heartbeat LED. Once this setting is set, it will persist between reboots. Note, this only comes into effect on the next boot, it does not stop the already running heartbeat.

pycom.rgbled(color)

Set the colour of the RGB LED. The colour is specified as 24 bit value representing red, green and blue, where the red colour is represented by the 8 most significant bits. For instance, passing the value `0x00FF00` will light up the LED in a very bright green.

pycom.nvs_set(key, value)

Set the value of the specified key in the NVRAM memory area of the external flash. Data stored here is preserved across resets and power cycles. Value can only take 32-bit integers at the moment. Example:

```
import pycom

pycom.nvs_set('temp', 25)
pycom.nvs_set('count', 10)
```

pycom.nvs_get(key)

Get the value the specified key from the NVRAM memory area of the external flash. Example:

```
import pycom

pulses = pycom.nvs_get('count')
```

If a non-existing key is given the returned value will be `None` .

pycom.nvs_erase(key)

Erase the given key from the NVRAM memory area.

pycom.nvs_erase_all()

Erase the entire NVRAM memory area.

pycom.wifi_on_boot([enable])

Get or set the WiFi on boot flag. When this flag is set to `True` , the AP with the default SSID (`lopy-wlan-xxx` for example) will be enabled as part of the boot process. If the flag is set to False, the module will boot with WiFi disabled until it's enabled by the script via the `WLAN` class. This setting is stored in non-volatile memory which preserves it across resets and power cycles. Example:

```
import pycom

pycom.wifi_on_boot(True)   # enable WiFi on boot

pycom.wifi_on_boot()       # get the wifi on boot flag
```

pycom.wdt_on_boot([enable])

Enables the WDT at boot time with the timeout in ms set by the function `wdt_on_boot_timeout` . If this flag is set, the application needs to reconfigure the WDT with a new timeout and feed it regularly to avoid a reset.

```
import pycom

pycom.wdt_on_boot(True)     # enable WDT on boot

pycom.wdt_on_boot()         # get the WDT on boot flag
```

pycom.wdt_on_boot_timeout([timeout])

Sets or gets the WDT on boot timeout in milliseconds. The minimum value is 5000 ms.

```
import pycom

pycom.wdt_on_boot_timeout(10000)     # set the timeout to 5000ms

pycom.wdt_on_boot_timeout()         # get the WDT timeout value
```

pycom.pulses_get(pin, timeout)

Return a list of pulses at `pin`. The methods scans for transitions at `pin` and returns a list of tuples, each telling the pin value and the duration in microseconds of that value. `pin` is a pin object, which must have set to `INP` or `OPEN_DRAIN` mode. The scan stops if not transitions occurs within `timeout` milliseconds.

Example:

```
# get the raw data from a DHT11/DHT22/AM2302 sensor
from machine import Pin
from pycom import pulses_get
from time import sleep_ms

pin = Pin("G7", mode=Pin.OPEN_DRAIN)
pin(0)
sleep_ms(20)
pin(1)
data = pulses_get(pin, 100)
```

pycom.ota_start()

pycom.ota_write(buffer)

pycom.ota_finish()

Perform a firmware update. These methods are internally used by a firmware update though FTP. The update starts with a call to `ota_start()`, followed by a series of calls to `ota_write(buffer)`, and is terminated with `ota_finish()`. After reset, the new image gets active. `buffer` shall hold the image data to be written, in arbitrary sizes. A block size of 4096 is recommended.

Example:

```
# Firmware update by reading the image from the SD card
#
from pycom import ota_start, ota_write, ota_finish
from os import mount
from machine import SD

BLOCKSIZE = const(4096)
APPIMG = "/sd/appimg.bin"

sd = SD()
mount(sd, '/sd')

with open(APPIMG, "rb") as f:
    buffer = bytearray(BLOCKSIZE)
    mv = memoryview(buffer)
    size=0
    ota_start()
    while True:
        chunk = f.readinto(buffer)
        if chunk > 0:
            ota_write(mv[:chunk])
            size += chunk
            print("\r%7d " % size, end="")
        else:
            break
    ota_finish()
```

Instead of reading the data to be written from a file, it can obviously also be received from a server using any suitable protocol, without the need to store it in the devices file system.

Micropython libraries

The following list contains the standard Python libraries, MicroPython-specific libraries and Pycom specific modules that are available on the Pycom devices.

The standard Python libraries have been "micro-ified" to fit in with the philosophy of MicroPython. They provide the core functionality of that module and are intended to be a drop-in replacement for the standard Python library.

> Some modules are available by an u-name, and also by their non-u-name. The non-u-name can be overridden by a file of that name in your package path. For example, `import json` will first search for a file `json.py` or directory `json` and load that package if it's found. If nothing is found, it will fallback to loading the built-in `ujson` module.

class micropython – MicroPython Internals Controls

Functions

micropython.alloc_emergency_exception_buf(size)

Allocate size bytes of RAM for the emergency exception buffer (a good size is around 100 bytes). The buffer is used to create exceptions in cases when normal RAM allocation would fail (eg within an interrupt handler) and therefore give useful traceback information in these situations.

A good way to use this function is to place it at the start of a main script (e.g. `boot.py` or `main.py`) and then the emergency exception buffer will be active for all the code following it.

micropython.const(expr)

Used to declare that the expression is a constant so that the compile can optimise it. The use of this function should be as follows:

```
from micropython import const

CONST_X = const(123)
CONST_Y = const(2 * CONST_X + 1)
```

Constants declared this way are still accessible as global variables from outside the module they are declared in. On the other hand, if a constant begins with an underscore then it is hidden, it is not available as a global variable, and does not take up any memory during execution.

This const function is recognised directly by the MicroPython parser and is provided as part of the `micropython` module mainly so that scripts can be written which run under both CPython and MicroPython, by following the above pattern.

micropython.opt_level([level])

If `level` is given then this function sets the optimisation level for subsequent compilation of scripts, and returns `None`. Otherwise it returns the current optimisation level.

micropython.mem_info([verbose])

Print information about currently used memory. If the `verbose` argument is given then extra information is printed.

The information that is printed is implementation dependent, but currently includes the amount of stack and heap used. In verbose mode it prints out the entire heap indicating which blocks are used and which are free.

micropython.qstr_info([verbose])

Print information about currently interned strings. If the `verbose` argument is given then extra information is printed.

The information that is printed is implementation dependent, but currently includes the number of interned strings and the amount of RAM they use. In verbose mode it prints out the names of all RAM-interned strings.

micropython.stack_use()

Return an integer representing the current amount of stack that is being used. The absolute value of this is not particularly useful, rather it should be used to compute differences in stack usage at different points.

micropython.heap_lock()

micropython.heap_unlock()

Lock or unlock the heap. When locked no memory allocation can occur and a `MemoryError` will be raised if any heap allocation is attempted.

These functions can be nested, i.e. `heap_lock()` can be called multiple times in a row and the lock-depth will increase, and then `heap_unlock()` must be called the same number of times to make the heap available again.

micropython.kbd_intr(chr)

Set the character that will raise a `KeyboardInterrupt` exception. By default this is set to 3 during script execution, corresponding to `Ctrl-C`. Passing `-1` to this function will disable capture of `Ctrl-C`, and passing `3` will restore it.

This function can be used to prevent the capturing of `Ctrl-C` on the incoming stream of characters that is usually used for the REPL, in case that stream is used for other purposes.

uctypes – Access Binary Data in a Structured Format

This module implements "foreign data interface" for MicroPython. The idea behind it is similar to CPython's `ctypes` modules, but the actual API is different, streamlined and optimised for small size. The basic idea of the module is to define data structure layout with about the same power as the C language allows, and the access it using familiar dot-syntax to reference sub-fields.

> Module ustruct Standard Python way to access binary data structures (doesn't scale well to large and complex structures).

Defining Structure Layout

Structure layout is defined by a "descriptor" - a Python dictionary which encodes field names as keys and other properties required to access them as associated values. Currently, `uctypes` requires explicit specification of offsets for each field. Offset are given in bytes from a structure start.

Following are encoding examples for various field types:

- Scalar types:

```
"field_name": uctypes.UINT32 | 0
```

In other words, value is scalar type identifier OR-ed with field offset (in bytes) from the start of the structure.

- Recursive structures:

```
"sub": (2, {
    "b0": uctypes.UINT8 | 0,
    "b1": uctypes.UINT8 | 1,
})
```

I.e. value is a 2-tuple, first element of which is offset, and second is a structure descriptor dictionary (note: offsets in recursive descriptors are relative to a structure it defines).

- Arrays of Primitive Types:

```
"arr": (uctypes.ARRAY | 0, uctypes.UINT8 | 2),
```

I.e. value is a 2-tuple, first element of which is ARRAY flag OR-ed with offset, and second is scalar element type OR-ed number of elements in array.

- Arrays of Aggregate Types:

```
"arr2": (uctypes.ARRAY | 0, 2, {"b": uctypes.UINT8 | 0}),
```

I.e. value is a 3-tuple, first element of which is ARRAY flag OR-ed with offset, second is a number of elements in array, and third is descriptor of element type.

- Pointer to a primitive type:

```
"ptr": (uctypes.PTR | 0, uctypes.UINT8),
```

I.e. value is a 2-tuple, first element of which is PTR flag OR-ed with offset, and second is scalar element type.

- Pointer to an aggregate type:

```
"ptr2": (uctypes.PTR | 0, {"b": uctypes.UINT8 | 0}),
```

I.e. value is a 2-tuple, first element of which is PTR flag OR-ed with offset, second is descriptor of type pointed to.

- Bitfields:

```
"bitf0": uctypes.BFUINT16 | 0 | 0 << uctypes.BF_POS | 8 << uctypes.BF_LEN,
```

I.e. value is type of scalar value containing given bitfield (typenames are similar to scalar types, but prefixes with "BF"), OR-ed with offset for scalar value containing the bitfield, and further OR-ed with values for bit offset and bit length of the bitfield within scalar value, shifted by BF_POS and BF_LEN positions, respectively. Bitfield position is counted from the least significant bit, and is the number of right-most bit of a field (in other words, it's a number of bits a scalar needs to be shifted right to extra the bitfield).

In the example above, first `UINT16` value will be extracted at offset 0 (this detail may be important when accessing hardware registers, where particular access size and alignment are required), and then bitfield whose rightmost bit is least-significant bit of this `UINT16`, and length is 8 bits, will be extracted - effectively, this will access least-significant byte of `UINT16`.

Note that bitfield operations are independent of target byte endianness, in particular, example above will access least-significant byte of `UINT16` in both little- and big-endian structures. But it depends on the least significant bit being numbered 0. Some targets may use different numbering in their native ABI, but `uctypes` always uses normalised numbering described above.

Module Contents

class **uctypes.struct(addr, descriptor, layout_type=NATIVE)**

Instantiate a "foreign data structure" object based on structure address in memory, descriptor (encoded as a dictionary), and layout type (see below).

uctypes.LITTLE_ENDIAN

Layout type for a little-endian packed structure. (Packed means that every field occupies exactly as many bytes as defined in the descriptor, i.e. the alignment is 1).

uctypes.BIG_ENDIAN

Layout type for a big-endian packed structure.

uctypes.NATIVE

Layout type for a native structure - with data endianness and alignment conforming to the ABI of the system on which MicroPython runs.

uctypes.sizeof(struct)

Return size of data structure in bytes. Argument can be either structure class or specific instantiated structure object (or its aggregate field).

uctypes.addressof(obj)

Return address of an object. Argument should be bytes, `bytearray` or other object supporting buffer protocol (and address of this buffer is what actually returned).

uctypes.bytes_at(addr, size)

Capture memory at the given address and size as bytes object. As bytes object is immutable, memory is actually duplicated and copied into bytes object, so if memory contents change later, created object retains original value.

uctypes.bytearray_at(addr, size)

Capture memory at the given address and size as `bytearray` object. Unlike `bytes_at()` function above, memory is captured by reference, so it can be both written too, and you will access current value at the given memory address.

Structure Descriptors and Instantiating Structure Objects

Given a structure descriptor dictionary and its layout type, you can instantiate a specific structure instance at a given memory address using uctypes.struct() constructor. Memory address usually comes from following sources:

- Predefined address, when accessing hardware registers on a baremetal system. Lookup these addresses in datasheet for a particular MCU/SoC.
- As a return value from a call to some FFI (Foreign Function Interface) function.
- From uctypes.addressof(), when you want to pass arguments to an FFI function, or alternatively, to access some data for I/O (for example, data read from a file or network socket).

Structure objects

Structure objects allow accessing individual fields using standard dot notation: `my_struct.substruct1.field1`. If a field is of scalar type, getting it will produce a primitive value (Python integer or float) corresponding to the value contained in a field. A scalar field can also be assigned to.

If a field is an array, its individual elements can be accessed with the standard subscript operator `[]` - both read and assigned to.

If a field is a pointer, it can be dereferenced using `[0]` syntax (corresponding to C `*` operator, though `[0]` works in C too). Subscripting a pointer with other integer values but 0 are supported too, with the same semantics as in C.

Summing up, accessing structure fields generally follows C syntax, except for pointer dereference, when you need to use `[0]` operator instead of `*`.

Limitations

Accessing non-scalar fields leads to allocation of intermediate objects to represent them. This means that special care should be taken to layout a structure which needs to be accessed when memory allocation is disabled (e.g. from an interrupt). The recommendations are:

- Avoid nested structures. For example, instead of `mcu_registers.peripheral_a.register1`, define separate layout descriptors for each peripheral, to be accessed as `peripheral_a.register1`.
- Avoid other non-scalar data, like array. For example, instead of `peripheral_a.register[0]` use `peripheral_a.register0`.

Note that these recommendations will lead to decreased readability and conciseness of layouts, so they should be used only if the need to access structure fields without allocation is anticipated (it’s even possible to define 2 parallel layouts - one for normal usage, and a restricted one to use when memory allocation is prohibited).

sys – System Specific Functions

Functions

sys.exit(retval=0)

Terminate current program with a given exit code. Underlyingly, this function raise as `SystemExit` exception. If an argument is given, its value given as an argument to `SystemExit`.

sys.print_exception(exc, file=sys.stdout)

Print exception with a traceback to a file-like object file (or `sys.stdout` by default).

> Difference to CPython
>
> This is simplified version of a function which appears in the traceback module in CPython. Unlike `traceback.print_exception()`, this function takes just exception value instead of exception type, exception value, and traceback object; file argument should be positional; further arguments are not supported. CPython-compatible traceback module can be found in `micropython-lib`.

Constants

sys.argv
A mutable list of arguments the current program was started with.

sys.byteorder
The byte order of the system ("little" or "big").

sys.implementation
Object with information about the current Python implementation. For MicroPython, it has following attributes:

- *name* - string "micropython"
- *version* - tuple (major, minor, micro), e.g. (1, 7, 0) This object is the recommended way to distinguish MicroPython from other Python implementations (note that it still may not exist in the very minimal ports).

> Difference to CPython

CPython mandates more attributes for this object, but the actual useful bare minimum is implemented in MicroPython.

sys.maxsize
Maximum value which a native integer type can hold on the current platform, or maximum value representable by MicroPython integer type, if it's smaller than platform max value (that is the case for MicroPython ports without long int support).

This attribute is useful for detecting "bitness" of a platform (32-bit vs 64-bit, etc.). It's recommended to not compare this attribute to some value directly, but instead count number of bits in it:

```
bits = 0
v = sys.maxsize
while v:
    bits += 1
    v >>= 1
if bits > 32:
    # 64-bit (or more) platform
else:
    # 32-bit (or less) platform
    # Note that on 32-bit platform, value of bits may be less than 32
    # (e.g. 31) due to peculiarities described above, so use "> 16",
    # "> 32", "> 64" style of comparisons.
```

sys.modules
Dictionary of loaded modules. On some ports, it may not include builtin modules.

sys.path
A mutable list of directories to search for imported modules.

sys.platform
The platform that MicroPython is running on. For OS/RTOS ports, this is usually an identifier of the OS, e.g. `linux`. For baremetal ports, it is an identifier of a board, e.g. `pyboard` for the original MicroPython reference board. It thus can be used to distinguish one board from another. If you need to check whether your program runs on MicroPython (vs other Python implementation), use `sys.implementation` instead.

sys.stderr
Standard error stream.

sys.stdin
Standard input stream.

sys.stdout
Standard output stream.

sys.version
Python language version that this implementation conforms to, as a string.

sys.version_info
Python language version that this implementation conforms to, as a tuple of ints.

uos – Basic "Operating System" Services

The `uos` module contains functions for filesystem access and `urandom` function.

Port Specifics

The filesystem has `/` as the root directory and the available physical drives are accessible from here. They are currently:

- `/flash` – the internal flash filesystem
- `/sd` – the SD card (if it exists)

Functions

uos.uname()

Return information about the system, firmware release version, and MicroPython interpreter version.

uos.chdir(path)

Change current directory.

uos.getcwd()

Get the current directory.

uos.listdir([dir])

With no argument, list the current directory. Otherwise list the given directory.

uos.mkdir(path)

Create a new directory.

uos.remove(path)

Remove a file.

uos.rmdir(path)

Remove a directory.

uos.rename(old_path, new_path)

Rename a file.

uos.stat(path)

Get the status of a file or directory.

The return value is a tuple with the following 10 values, in order:

- `st_mode` : protection bits.
- `st_ino` : `inode` number. (not implemented, returns 0)
- `st_dev` : device. (not implemented, returns 0)
- `st_nlink` : number of hard links. (not implemented, returns 0)
- `st_uid` : user id of owner. (not implemented, returns 0)
- `st_gid` : group id of owner. (not implemented, returns 0)
- `st_size` : size of file in bytes.
- `st_atime` : time of most recent access.
- `st_mtime` : time of most recent content modification.
- `st_ctime` : time of most recent metadata change.

uos.getfree(path)

Returns the free space (in KiB) in the drive specified by path.

uos.sync()

Sync all filesystems.

uos.urandom(n)

Return a bytes object with n random bytes.

uos.unlink(path)

Alias for the `remove()` method.

uos.mount(block_device, mount_point, * , readonly=False)

Mounts a block device (like an SD object) in the specified mount point. Example:

```
os.mount(sd, '/sd')
uos.unmount(path)
```

Unmounts a previously mounted block device from the given path.

uos.mkfs(block_device or path)

Formats the specified path, must be either `/flash` or `/sd`. A block device can also be passed like an SD object before being mounted.

uos.dupterm(stream_object)

Duplicate the terminal (the REPL) on the passed stream-like object. The given object must at least implement the `read()` and `write()` methods.

Constants

uos.sep
Separation character used in paths

array – Arrays of Numeric Data

See Python array for more information.

Supported format codes: `b, B, h, H, i, I, l, L, q, Q, f, d` (the latter 2 depending on the floating-point support).

Classes

class array.array(typecode[, iterable])

Create array with elements of given type. Initial contents of the array are given by an iterable. If it is not provided, an empty array is created.

array.append(val)

Append new element to the end of array, growing it.

array.extend(iterable)

Append new elements as contained in an iterable to the end of array, growing it.

cmath – Mathematical Functions for Complex Numbers

The `cmath` module provides some basic mathematical functions for working with complex numbers. Floating point support required for this module.

Functions

cmath.cos(z)

Return the cosine of `z`.

cmath.exp(z)

Return the exponential of `z`.

cmath.log(z)

Return the natural logarithm of `z`. The branch cut is along the negative real axis.

cmath.log10(z)

Return the base-10 logarithm of `z`. The branch cut is along the negative real axis.

cmath.phase(z)

Returns the phase of the number `z`, in the range (-pi, +pi).

cmath.polar(z)

Returns, as a tuple, the polar form of `z`.

cmath.rect(r, phi)

Returns the complex number with modulus `r` and phase `phi`.

cmath.sin(z)

Return the sine of `z`.

cmath.sqrt(z)

Return the square-root of `z`.

Constants

cmath.e
Base of the natural logarithm

cmath.pi
The ratio of a circle’s circumference to its diameter

math – Mathematical Functions

The math module provides some basic mathematical functions for working with floating-point numbers. Floating point support required for this module.

Functions

math.acos(x)

Return the inverse cosine of `x`.

math.acosh(x)

Return the inverse hyperbolic cosine of `x`.

math.asin(x)

Return the inverse sine of `x`.

math.asinh(x)

Return the inverse hyperbolic sine of `x`.

math.atan(x)

Return the inverse tangent of `x`.

math.atan2(y, x)

Return the principal value of the inverse tangent of `y/x`.

math.atanh(x)

Return the inverse hyperbolic tangent of `x`.

math.ceil(x)

Return an integer, being x rounded towards positive infinity.

math.copysign(x, y)

Return x with the sign of `y`.

math.cos(x)

Return the cosine of `x` .

math.cosh(x)

Return the hyperbolic cosine of `x` .

math.degrees(x)

Return radians `x` converted to degrees.

math.erf(x)

Return the error function of `x` .

math.erfc(x)

Return the complementary error function of `x` .

math.exp(x)

Return the exponential of `x` .

math.expm1(x)

Return `exp(x) - 1` .

math.fabs(x)

Return the absolute value of `x` .

math.floor(x)

Return an integer, being `x` rounded towards negative infinity.

math.fmod(x, y)

Return the remainder of `x/y` .

math.frexp(x)

Decomposes a floating-point number into its mantissa and exponent. The returned value is the tuple `(m, e)` such that `x == m * 2**e` exactly. If `x == 0` then the function returns `(0.0, 0)` , otherwise the relation `0.5 <= abs(m) < 1` holds.

math.gamma(x)

Return the gamma function of `x` .

math.isfinite(x)

Return `True` if `x` is finite.

math.isinf(x)

Return `True` if `x` is infinite.

math.isnan(x)

Return `True` if `x` is not-a-number

math.ldexp(x, exp)

Return `x * (2**exp)` .

math.lgamma(x)

Return the natural logarithm of the gamma function of `x` .

math.log(x)

Return the natural logarithm of `x` .

math.log10(x)

Return the base-10 logarithm of `x` .

math.log2(x)

Return the base-2 logarithm of `x` .

math.modf(x)

Return a tuple of two floats, being the fractional and integral parts of `x` . Both return values have the same sign as `x` .

math.pow(x, y)

Returns `x` to the power of `y` .

math.radians(x)

Return degrees `x` converted to radians.

math.sin(x)

Return the sine of `x` .

math.sinh(x)

Return the hyperbolic sine of `x`.

math.sqrt(x)

Return the square root of `x`.

math.tan(x)

Return the tangent of `x`.

math.tanh(x)

Return the hyperbolic tangent of `x`.

math.trunc(x)

Return an integer, being `x` rounded towards `0`.

Constants

math.e
Base of the natural logarithm

math.pi
The ratio of a circle’s circumference to its diameter

gc – Garbage Collector

Functions

gc.enable()

Enable automatic garbage collection.

gc.disable()

Disable automatic garbage collection. Heap memory can still be allocated, and garbage collection can still be initiated manually using gc.collect().

gc.collect()

Run a garbage collection.

gc.mem_alloc()

Return the number of bytes of heap RAM that are allocated.

gc.mem_free()

Return the number of bytes of available heap RAM.

ubinascii – Binary/ASCII Conversions

This module implements conversions between binary data and various encodings of it in ASCII form (in both directions).

Functions

ubinascii.hexlify(data[, sep])

Convert binary data to hexadecimal representation. Returns bytes string.

> Difference to CPython
>
> If additional argument, `sep` is supplied, it is used as a separator between hexadecimal values.

ubinascii.unhexlify(data)

Convert hexadecimal data to binary representation. Returns bytes string. (i.e. inverse of `hexlify`)

ubinascii.a2b_base64(data)

Convert Base64-encoded data to binary representation. Returns bytes string.

ubinascii.b2a_base64(data)

Encode binary data in Base64 format. Returns string.

ujson – JSON Encoding and Decoding

This modules allows to convert between Python objects and the JSON data format.

Functions

ujson.dumps(obj)

Return `obj` represented as a JSON string.

ujson.loads(str)

Parse the JSON `str` and return an object. Raises `ValueError` if the string is not correctly formed.

ujson.load(fp)

Parse contents of `fp` (a `.read()` -supporting file-like object containing a JSON document). Raises `ValueError` if the content is not correctly formed.

ure – regular expressions

This module implements regular expression operations. Regular expression syntax supported is a subset of CPython re module (and actually is a subset of POSIX extended regular expressions).

Supported operators are:

`.` Match any character. `[]` Match set of characters. Individual characters and ranges are supported.

```
^
$
?
*
+
??
*?
+?
```

Counted repetitions `({m,n})`, more advanced assertions, named groups, etc. are not supported.

Functions

ure.compile(regex)

Compile regular expression, return `regex object`.

ure.match(regex, string)

Match regex against `string`. Match always happens from starting position in a string.

ure.search(regex, string)

Search regex in a string. Unlike match, this will search string for first position which matches regex (which still may be 0 if regex is anchored).

ure.DEBUG

Flag value, display debug information about compiled expression.

Regex objects

Compiled regular expression. Instances of this class are created using `ure.compile()` .

regex.match(string)

regex.search(string)

regex.split(string, max_split=-1)

Match objects

Match objects as returned by `match()` and `search()` methods.

match.group([index])

Only numeric groups are supported.

usocket – Socket Module

This module provides access to the BSD socket interface.

See corresponding CPython module for comparison.

Socket Address Format(s)

Functions below which expect a network address, accept it in the format of `(ipv4_address, port)`, where `ipv4_address` is a string with dot-notation numeric IPv4 address, e.g. `8.8.8.8`, and port is integer port number in the range 1-65535. Note the domain names are not accepted as `ipv4_address`, they should be resolved first using `socket.getaddrinfo()`.

Functions

socket.socket(socket.AF_INET, socket.SOCK_STREAM, socket.IPPROTO_TCP)

Create a new socket using the given address family, socket type and protocol number.

socket.getaddrinfo(host, port)

Translate the host/port argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service. The list of 5-tuples has following structure:

`(family, type, proto, canonname, sockaddr)` The following example shows how to connect to a given url:

```
s = socket.socket()
s.connect(socket.getaddrinfo('www.micropython.org', 80)[0][-1])
```

Exceptions

socket.error socket.timeout

Constants

socket.AF_INET socket.AF_LORA

Family types

socket.SOCK_STREAM socket.SOCK_DGRAM socket.SOCK_RAW

Socket types

socket.IPPROTO_UDP socket.IPPROTO_TCP

Socket protocols

socket.SOL_SOCKET socket.SOL_LORA socket.SOL_SIGFOX

Socket options layers

socket.SO_REUSEADDR

IP socket options

socket.SO_CONFIRMED socket.SO_DR

LoRa socket options

socket.SO_RX socket.SO_TX_REPEAT socket.SO_OOB socket.SO_BIT

Sigfox socket options

class Socket

Methods

socket.close()

Mark the socket closed. Once that happens, all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed).

Sockets are automatically closed when they are garbage-collected, but it is recommended to `close()` them explicitly, or to use a with statement around them.

socket.bind(address)

Bind the `socket` to `address`. The socket must not already be bound. The `address` parameter must be a tuple containing the IP address and the port.

> In the case of LoRa sockets, the address parameter is simply an integer with the port number, for instance: `s.bind(1)`

socket.listen([backlog])

Enable a server to accept connections. If backlog is specified, it must be at least 0 (if it's lower, it will be set to 0); and specifies the number of unaccepted connections that the system will allow before refusing new connections. If not specified, a default reasonable value is chosen.

socket.accept()

Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair `(conn, address)` where `conn` is a new socket object usable to send and receive data on the connection, and `address` is the address bound to the socket on the other end of the connection.

socket.connect(address)

Connect to a remote socket at `address`.

socket.send(bytes)

Send data to the socket. The socket must be connected to a remote socket.

socket.sendall(bytes)

Alias of `socket.send(bytes)`.

socket.recv(bufsize)

Receive data from the socket. The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by `bufsize`.

socket.sendto(bytes, address)

Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by address.

socket.recvfrom(bufsize)

Receive data from the socket. The return value is a pair `(bytes, address)` where `bytes` is a bytes object representing the data received and `address` is the address of the socket sending the data.

socket.setsockopt(level, optname, value)

Set the value of the given socket option. The needed symbolic constants are defined in the socket module (`SO_*` etc.). The value can be an integer or a bytes-like object representing a buffer.

socket.settimeout(value)

Set a timeout on blocking socket operations. The value argument can be a nonnegative floating point number expressing seconds, or `None`. If a non-zero value is given, subsequent socket operations will raise a timeout exception if the timeout period value has elapsed before the operation has completed. If zero is given, the socket is put in non-blocking mode. If None is given, the socket is put in blocking mode.

socket.setblocking(flag)

Set blocking or non-blocking mode of the socket: if flag is false, the socket is set to non-blocking, else to blocking mode.

This method is a shorthand for certain `settimeout()` calls:

```
sock.setblocking(True) is equivalent to sock.settimeout(None)
sock.setblocking(False) is equivalent to sock.settimeout(0.0)
```

socket.makefile(mode='rb')

Return a file object associated with the socket. The exact returned type depends on the arguments given to makefile(). The support is limited to binary modes only (`rb` and `wb`). CPython's arguments: `encoding` , `errors` , and `newline` are not supported.

The socket must be in blocking mode; it can have a timeout, but the file object's internal buffer may end up in a inconsistent state if a timeout occurs.

> Difference to CPython
>
> Closing the file object returned by `makefile()` WILL close the original socket as well.

socket.read(size)

Read up to size bytes from the socket. Return a bytes object. If `size` is not given, it behaves just like socket.readall(), see below.

socket.readall()

Read all data available from the socket until EOF. This function will not return until the socket is closed.

socket.readinto(buf[, nbytes])

Read bytes into the `buf`. If `nbytes` is specified then read at most that many bytes. Otherwise, read at most `len(buf)` bytes.

Return value: number of bytes read and stored into `buf`.

socket.readline()

Read a line, ending in a newline character.

Return value: the line read.

socket.write(buf)

Write the buffer of bytes to the socket.

Return value: number of bytes written.

select – Wait for Events on a Set of Streams

This module provides functions to wait for events on streams (select streams which are ready for operations).

Pyboard specifics

Polling is an efficient way of waiting for read/write activity on multiple objects. Current objects that support polling are: pyb.UART, pyb.USB_VCP.

Functions

select.poll()

Create an instance of the `Poll` class.

select.select(rlist, wlist, xlist[, timeout])

Wait for activity on a set of objects.

This function is provided for compatibility and is not efficient. Usage of `Poll` is recommended instead.

***class* Poll**

Methods

poll.register(obj[, eventmask])

Register `obj` for polling. `eventmask` is logical OR of:

- `select.POLLIN` - data available for reading
- `select.POLLOUT` - more data can be written
- `select.POLLERR` - error occurred
- `select.POLLHUP` - end of stream/connection termination detected `eventmask` defaults to `select.POLLIN | select.POLLOUT`.

poll.unregister(obj)

Unregister `obj` from polling.

poll.modify(obj, eventmask)

Modify the `eventmask` for `obj`.

poll.poll([timeout])

Wait for at least one of the registered objects to become ready. Returns list of (`obj`, `event`, ...) tuples, `event` element specifies which events happened with a stream and is a combination of `select.POLL*` constants described above. There may be other elements in tuple, depending on a platform and version, so don't assume that its size is 2. In case of timeout, an empty list is returned.

Timeout is in milliseconds.

utime – Time Functions

The `utime` module provides functions for getting the current time and date, measuring time intervals, and for delays.

Time Epoch: Pycom's ESP32 port uses standard for POSIX systems epoch of `1970-01-01 00:00:00 UTC`.

Maintaining actual calendar date/time

This requires a Real Time Clock (RTC). On systems with underlying OS (including some RTOS), an RTC may be implicit. Setting and maintaining actual calendar time is responsibility of OS/RTOS and is done outside of MicroPython, it just uses OS API to query date/time. On baremetal ports however system time depends on `machine.RTC()` object. The current calendar time may be set using `machine.RTC().datetime(tuple)` function, and maintained by following means:

- By a backup battery (which may be an additional, optional component for a particular board).
- Using networked time protocol (requires setup by a port/user).
- Set manually by a user on each power-up (many boards then maintain RTC time across hard resets, though some may require setting it again in such case).

If actual calendar time is not maintained with a system/MicroPython RTC, functions below which require reference to current absolute time may behave not as expected.

Functions

utime.gmtime([secs])

Convert a time expressed in seconds since the Epoch (see above) into an 8-tuple which contains: `(year, month, mday, hour, minute, second, weekday, yearday)` If `secs` is not provided or `None`, then the current time from the RTC is used.

- `year` includes the century (for example 2014).
- `month` is 1-12
- `mday` is 1-31
- `hour` is 0-23
- `minute` is 0-59
- `second` is 0-59
- `weekday` is 0-6 for Mon-Sun

- `yearday` is 1-366

utime.localtime([secs])

Like `gmtime()` but converts to local time. If `secs` is not provided or `None`, the current time from the RTC is used.

utime.mktime()

This is inverse function of `localtime`. It's argument is a full 8-tuple which expresses a time as per `localtime`. It returns an integer which is the number of seconds since `Jan 1, 2000`.

utime.sleep(seconds)

Sleep for the given number of `seconds`. `seconds` can be a floating-point number to sleep for a fractional number of seconds. Note that other MicroPython ports may not accept floating-point argument, for compatibility with them use `sleep_ms()` and `sleep_us()` functions.

utime.sleep_ms(ms)

Delay for given number of milliseconds, should be positive or 0.

utime.sleep_us(us)

Delay for given number of microseconds, should be positive or 0

utime.ticks_ms()

Returns uptime, in milliseconds.

utime.ticks_us()

Just like `ticks_ms` above, but in microseconds.

utime.ticks_cpu()

Same as `ticks_us`, but faster.

utime.ticks_diff(old, new)

Measure period between consecutive calls to `ticks_ms()`, `ticks_us()`, or `ticks_cpu()`. The value returned by these functions may wrap around at any time, so directly subtracting them is not supported. `ticks_diff()` should be used instead. "old" value should actually precede "new" value in time, or result is undefined. This function should not be used to measure arbitrarily long periods of time (because `ticks_*()` functions wrap around and usually would have short period). The expected usage pattern is implementing event polling with timeout:

```
# Wait for GPIO pin to be asserted, but at most 500us
start = time.ticks_us()
while pin.value() == 0:
    if time.ticks_diff(start, time.ticks_us()) > 500:
        raise TimeoutError
```

utime.time()

Returns the number of seconds, as an integer, since the Epoch, assuming that underlying RTC is set. If an RTC is not set, this function returns number of seconds since power up or reset). If you want to develop portable MicroPython application, you should not rely on this function to provide higher than second precision. If you need higher precision, use `ticks_ms()` and `ticks_us()` functions, if you need calendar time, `localtime()` without an argument is a better choice.

utime.timezone([secs])

Set or get the timezone offset, in seconds. If `secs` is not provided, it returns the current value.

> In MicroPython, `time.timezone` works the opposite way to Python. In Python, to get the local time, you write `local_time = utc - timezone`, while in MicroPython it is `local_time = utc + timezone`.

uhashlib – Hashing Algorithm

This module implements binary data hashing algorithms. MD5 and SHA are supported. By limitations in the hardware, only one active hashing operation is supported at a time.

Constructors

class uhashlib.md5([data])

Create a MD5 hasher object and optionally feed data into it.

class uhashlib.sha1([data])

Create a SHA-1 hasher object and optionally feed data into it.

class uhashlib.sha224([data])

Create a SHA-224 hasher object and optionally feed data into it.

class uhashlib.sha256([data])

Create a SHA-256 hasher object and optionally feed data into it.

class uhashlib.sha384([data])

Create a SHA-384 hasher object and optionally feed data into it.

class uhashlib.sha512([data])

Create a SHA-512 hasher object and optionally feed data into it.

Methods

hash.update(data)

Feed more binary data into hash.

hash.digest()

Return hash for all data passed through hash, as a bytes object. After this method is called, more data cannot be fed into hash any longer.

hash.hexdigest()

This method is NOT implemented. Use `ubinascii.hexlify(hash.digest())` to achieve a similar effect.

ussl – ssl module

This module provides access to Transport Layer Security (often known as "Secure Sockets Layer") encryption and peer authentication facilities for network sockets, both client-side and server-side.

Functions

ssl.wrap_socket(sock, keyfile=None, certfile=None, server_side=False, cert_reqs=CERT_NONE, ca_certs=None)

Takes an instance `sock` of `socket.socket`, and returns an instance of ssl.SSLSocket, a subtype of `socket.socket`, which wraps the underlying socket in an SSL context. Example:

```
import socket
import ssl
s = socket.socket()
ss = ssl.wrap_socket(s)
ss.connect(socket.getaddrinfo('www.google.com', 443)[0][-1])
```

Certificates must be used in order to validate the other side of the connection, and also to authenticate ourselves with the other end. Such certificates must be stored as files using the FTP server, and they must be placed in specific paths with specific names.

For instance, to connect to the Blynk servers using certificates, take the file `ca.pem` located in the `blynk` examples folder and put it in `/flash/cert/`. Then do:

```
import socket
import ssl
s = socket.socket()
ss = ssl.wrap_socket(s, cert_reqs=ssl.CERT_REQUIRED, ca_certs='/flash/cert/ca.pem')
ss.connect(socket.getaddrinfo('cloud.blynk.cc', 8441)[0][-1])
```

SSL sockets inherit all methods and from the standard sockets, see the `usocket` module.

Exceptions

ssl.SSLError

Constants

ssl.CERT_NONE ssl.CERT_OPTIONAL ssl.CERT_REQUIRED

Supported values in `cert_reqs`

ucrypto — Cryptography

This module provides native support for cryptographic algorithms. It's loosely based on PyCrypto.

Classes

- class AES - Advanced Encryption Standard

Methods

crypto.getrandbits(bits)

Returns a bytes object filled with random bits obtained from the hardware random number generator.

According to the **ESP32 Technical Reference Manual**, such bits "... can be used as the basis for cryptographical operations". "These true random numbers are generated based on the noise in the Wi-Fi/BT RF system. When Wi-Fi and BT are disabled, the random number generator will give out pseudo-random numbers."

The parameter `bits` is rounded upwards to the nearest multiple of 32 bits.

Cryptography is not a trivial business. Doing things the wrong way could quickly result in decreased or no security. Please document yourself in the subject if you are depending on encryption to secure important information.

ustruct – Pack and Unpack Primitive Data Types

See Python struct for more information.

Supported size/byte order prefixes: `@, <, >, !`.

Supported format codes: `b, B, h, H, i, I, l, L, q, Q, s, P, f, d` (the latter 2 depending on the floating-point support).

Functions

ustruct.calcsize(fmt)

Return the number of bytes needed to store the given `fmt`.

ustruct.pack(fmt, v1, v2, ...)

Pack the values `v1, v2, ...` according to the format string `fmt`. The return value is a bytes object encoding the values.

ustruct.pack_into(fmt, buffer, offset, v1, v2, ...)

Pack the values `v1, v2, ...` according to the format string `fmt` into a buffer starting at `offset`. `offset` may be negative to count from the end of buffer.

ustruct.unpack(fmt, data)

Unpack from the `data` according to the format string `fmt`. The return value is a tuple of the unpacked values.

ustruct.unpack_from(fmt, data, offset=0)

Unpack from the `data` starting at `offset` according to the format string `fmt`. `offset` may be negative to count from the end of buffer. The return value is a tuple of the unpacked values.

thread - Low-level Threading API

This module provides low-level primitives for working with multiple threads (also called light-weight processes or tasks) — multiple threads of control sharing their global data space. For synchronisation, simple locks (also called mutexes or binary semaphores) are provided.

When a thread specific error occurs a `RuntimeError` exception is raised.

Quick Usage Example

```
import _thread
import time

def th_func(delay, id):
    while True:
        time.sleep(delay)
        print('Running thread %d' % id)

for i in range(2):
    _thread.start_new_thread(th_func, (i + 1, i))
```

Functions

_thread.start_new_thread(function, args[, kwargs])

Start a new thread and return its identifier. The thread executes the function with the argument list args (which must be a tuple). The optional `kwargs` argument specifies a dictionary of keyword arguments. When the function returns, the thread silently exits. When the function terminates with an unhandled exception, a stack trace is printed and then the thread exits (but other threads continue to run).

_thread.exit()

Raise the `SystemExit` exception. When not caught, this will cause the thread to exit silently.

_thread.allocate_lock()

Return a new lock object. Methods of locks are described below. The lock is initially unlocked.

_thread.get_ident()

Return the `thread identifier` of the current thread. This is a nonzero integer. Its value has no direct meaning; it is intended as a magic cookie to be used e.g. to index a dictionary of thread-specific data. Thread identifiers may be recycled when a thread exits and another thread is created.

_thread.stack_size([size])

Return the thread stack size (in bytes) used when creating new threads. The optional size argument specifies the stack size to be used for subsequently created threads, and must be `0` (use platform or configured default) or a positive integer value of at least `4096` (4KiB). 4KiB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself.

Objects

_thread.LockType
This is the type of lock objects.

class Lock – used for synchronisation between threads

Methods

Lock objects have the following methods:

lock.acquire(waitflag=1, timeout=-1)

Without any optional argument, this method acquires the lock unconditionally, if necessary waiting until it is released by another thread (only one thread at a time can acquire a lock — that’s their reason for existence).

If the integer `waitflag` argument is present, the action depends on its value: if it is zero, the lock is only acquired if it can be acquired immediately without waiting, while if it is nonzero, the lock is acquired unconditionally as above.

If the floating-point timeout argument is present and positive, it specifies the maximum wait time in seconds before returning. A negative timeout argument specifies an unbounded wait. You cannot specify a timeout if `waitflag` is zero.

The return value is `True` if the lock is acquired successfully, `False` if not.

lock.release()

Releases the lock. The lock must have been acquired earlier, but not necessarily by the same thread.

lock.locked()

Return the status of the lock: `True` if it has been acquired by some thread, `False` if not.

In addition to these methods, lock objects can also be used via the with statement, e.g.:

```
import _thread

a_lock = _thread.allocate_lock()

with a_lock:
    print("a_lock is locked while this executes")
```

Builtin Functions

All builtin functions are described here. They are also available via builtins module.

abs()

all()

any()

bin()

class bool

***class* bytearray**

***class* bytes**

callable()

chr()

***class* method()**

compile()

***class* complex**

***class* dict**

dir()

divmod()

enumerate()

eval()

exec()

filter()

***class* float**

***class* frozenset**

getattr()

globals()

hasattr()

hash()

hex()

id()

input()

***class* int**

isinstance()

issubclass()

iter()

len()

***class* list**

locals()

map()

max()

***class* memoryview**

min()

next()

***class* object**

oct()

open()

ord()

pow()

print()

property()

range()

repr()

reversed()

round()

***class* set**

setattr()

sorted()

staticmethod()

***class* str**

sum()

super()

***class* tuple**

type()

zip()

Product Info pages

The follow pages contain all information relating to each product, for examples: pinouts, spec sheets, relevant examples and notes.

Development Modules

- WiPy 2.0
- WiPy 3.0
- SiPy
- LoPy
- LoPy4
- GPy
- FiPy

OEM modules

- W01
- L01
- L04
- G01
- L01/W01 Reference Board
- Universal Reference Board

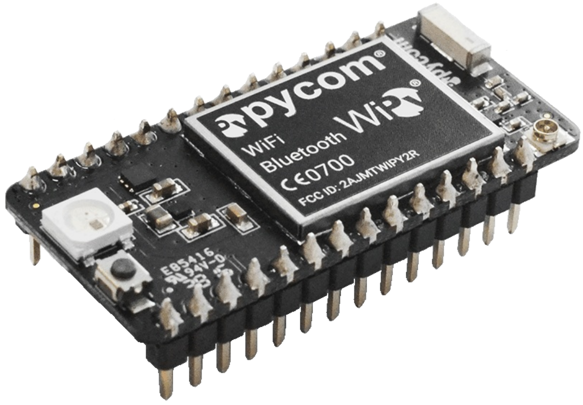
Shields and Expansion boards

- Expansion Board 2.0
- Pysense
- Pytrack
- Deep Sleep Shield

Development Devices

This section contains all of the datasheets for the Pycom Development Devices. This includes the WiPy 2.0 and 3.0, LoPy, LoPy 4, SiPy, GPy, and FiPy.

WiPy 2.0

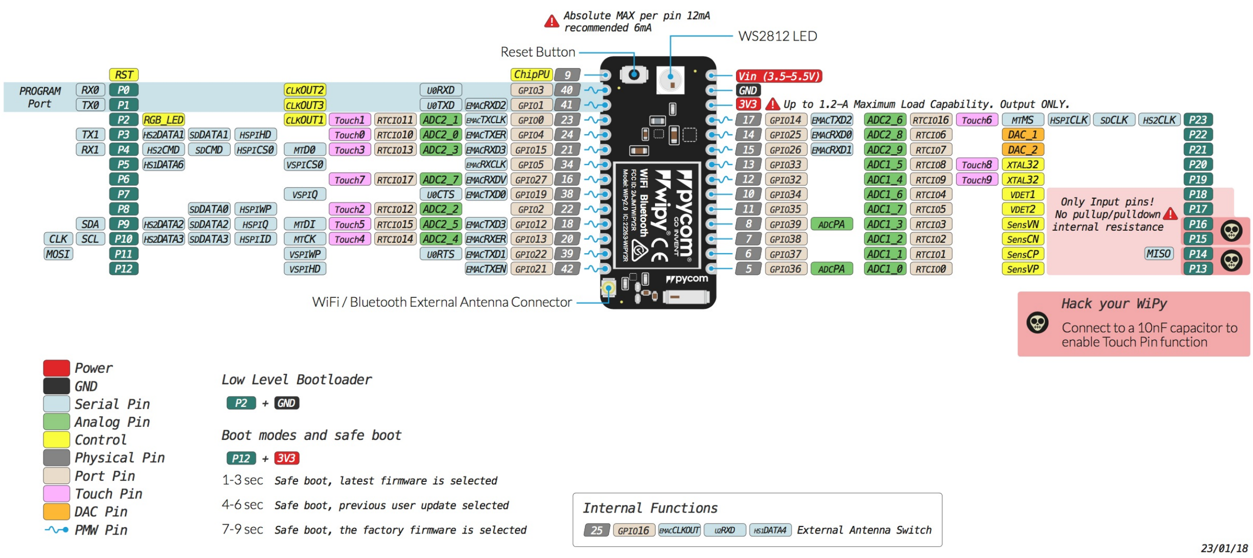


Store: Discontinued, See WiPy3

Getting Started Click Here

Pinout

The pinout of the WiPy2 is available as a PDF File.



Please note that the PIN assignments for UART1 (TX1/RX1), SPI (CLK,MOSI,MISO) and I2C (SDA,SCL) are defaults and can be changed in Software.

Datasheet

The datasheet of the WiPy2 is available as a PDF File.

Notes

WiFi

By default, upon boot the WiPy2 will create a WiFi access point with the SSID `wipy-wlan-xxxx` , where `xxxx` is a random 4-digit number, and the password `www.pycom.io` .

Power

The `Vin` pin on the WiPy2 can be supplied with a voltage ranging from `3.5v` to `5.5v` . The `3.3v` pin on the other hand is output **only**, and must not be used to feed power into the WiPy2, otherwise the on-board regulator will be damaged.

Deep Sleep

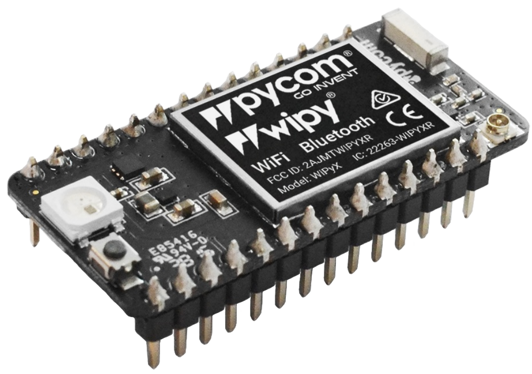
Due to a couple issues with the WiPy2 design the module draws more current than it should while in deep sleep. The DC-DC switching regulator always stays in high performance mode which is used to provide the lowest possible output ripple when the modules is in use. In this mode, it draws a quiescent current of 10mA. When the regulator is put into ECO mode, the quiescent current goes down to 10uA. Unfortunately, the pin used to control this mode is out of the RTC domain, and therefore not usable during deep sleep. This causes the regulator to always stay in PWM mode, keeping its quiescent current at 10mA. Alongside this the flash chip doesn't enter power down mode because the CS pin is floating during deep sleep. This causes the flash chip to consume around 2mA of current. To work around this issue a "deep sleep shield" is available that attaches to the module and allows power to be cut off from the device. The device can then be re-enabled either on a timer or via pin interrupt. With the deep sleep shield the current consumption during deep sleep is between 7uA and 10uA depending on the wake sources configured.

Tutorials

Tutorials on how to the WiPy2 module can be found in the examples section of this documentation. The following tutorials might be of specific interest for the WiPy2:

- WiFi connection
- BLE

WiPy 3.0

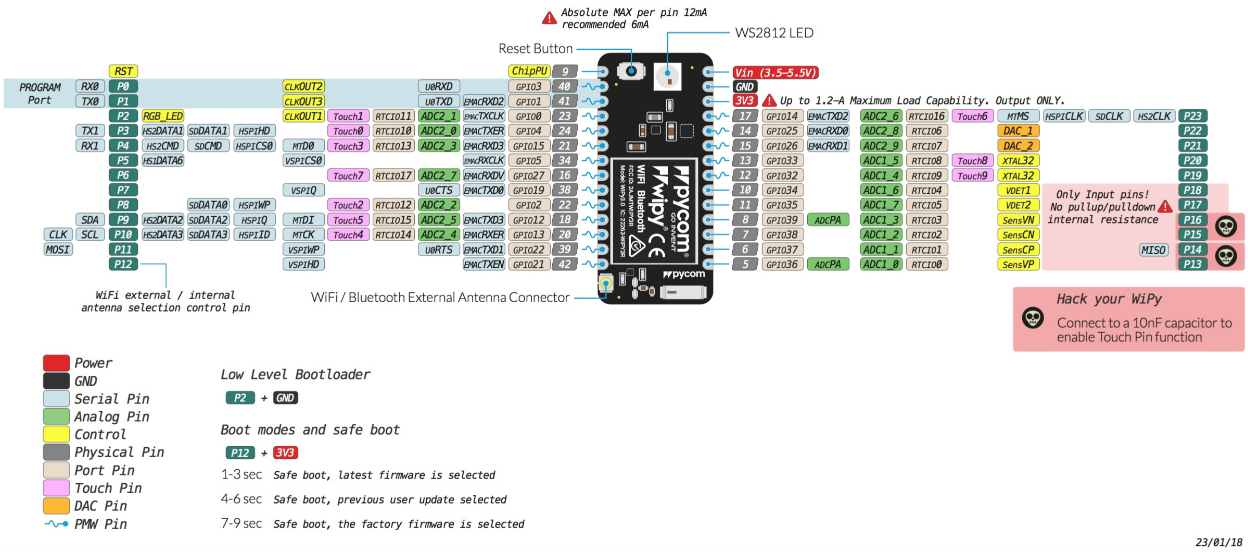


Store: Buy Here

Getting Started Click Here

Pinout

The pinout of the WiPy3 is available as a PDF File.



Please note that the PIN assignments for UART1 (TX1/RX1), SPI (CLK,MOSI,MISO) and I2C (SDA,SCL) are defaults and can be changed in Software.

Differences from WiPy 2.0

- Deep sleep current draw fixed, now only 19.7µA
- Upgraded RAM from 512KB to 4MB
- Upgraded External FLASH from 4MB to 8MB
- Antenna select pin moved from GPIO16 to GPIO21 (P12)

Datasheet

The datasheet of the WiPy3 is available as a PDF File.

Notes

WiFi

By default, upon boot the WiPy3 will create a WiFi access point with the SSID `wipy-wlan-xxxx` , where `xxxx` is a random 4-digit number, and the password `www.pycom.io` .

The RF switch that selects between the on-board and external antenna is connected to `P12` , for this reason using `P12` should be avoided unless WiFi is disabled in your application.

Power

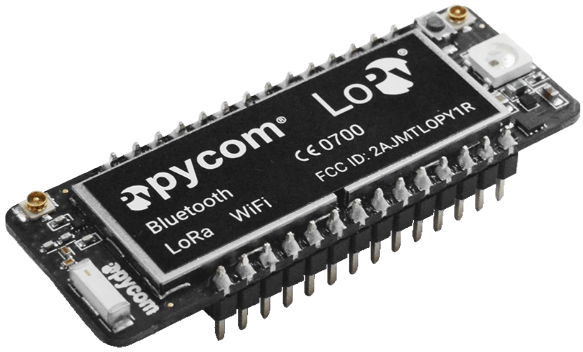
The `Vin` pin on the WiPy3 can be supplied with a voltage ranging from `3.5v` to `5.5v` . The `3.3v` pin on the other hand is output **only**, and must not be used to feed power into the WiPy3, otherwise the on-board regulator will be damaged.

Tutorials

Tutorials on how to the WiPy3 module can be found in the examples section of this documentation. The following tutorials might be of specific interest for the WiPy3:

- WiFi connection
- BLE

LoPy

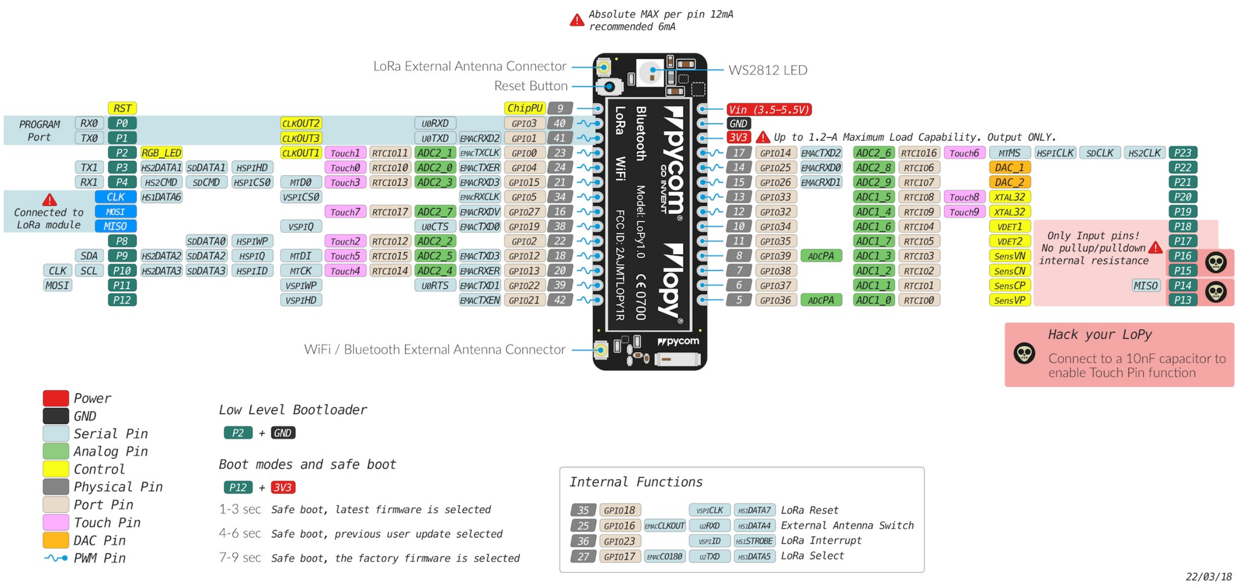


Store: Buy Here

Getting Started Click Here

Pinout

The pinout of the LoPy is available as a PDF File.



Please note that the PIN assignments for UART1 (TX1/RX1), SPI (CLK,MOSI,MISO) and I2C (SDA,SCL) are defaults and can be changed in Software.

Datasheet

The datasheet of the LoPy is available as a PDF File.

Notes

WiFi

By default, upon boot the LoPy will create a WiFi access point with the SSID `lopy-wlan-xxxx` , where `xxxx` is a random 4-digit number, and the password `www.pycom.io` .

Power

The `Vin` pin on the LoPy can be supplied with a voltage ranging from `3.5v` to `5.5v` . The `3.3v` pin on the other hand is output **only**, and must not be used to feed power into the LoPy, otherwise the on-board regulator will be damaged.

Deep Sleep

Due to a couple issues with the LoPy design the module draws more current than it should while in deep sleep. The DC-DC switching regulator always stays in high performance mode which is used to provide the lowest possible output ripple when the modules is in use. In this mode, it draws a quiescent current of 10mA. When the regulator is put into ECO mode, the quiescent current goes down to 10uA. Unfortunately, the pin used to control this mode is out of the RTC domain, and therefore not usable during deep sleep. This causes the regulator to always stay in PWM mode, keeping its quiescent current at 10mA. Alongside this the flash chip doesn't enter power down mode because the CS pin is floating during deep sleep. This causes the flash chip to consume around 2mA of current. To work around this issue a "deep sleep shield" is available that attaches to the module and allows power to be cut off from the device. The device can then be re-enabled either on a timer or via pin interrupt. With the deep sleep shield the current consumption during deep sleep is between 7uA and 10uA depending on the wake sources configured.

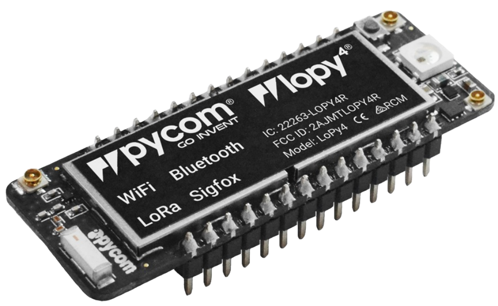
Tutorials

Tutorials on how to the LoPy module can be found in the examples section of this documentation. The following tutorials might be of specific interest for the LoPy:

- WiFi connection
- LoRaWAN node

- LoRaWAN nano gateway
- BLE

LoPy4

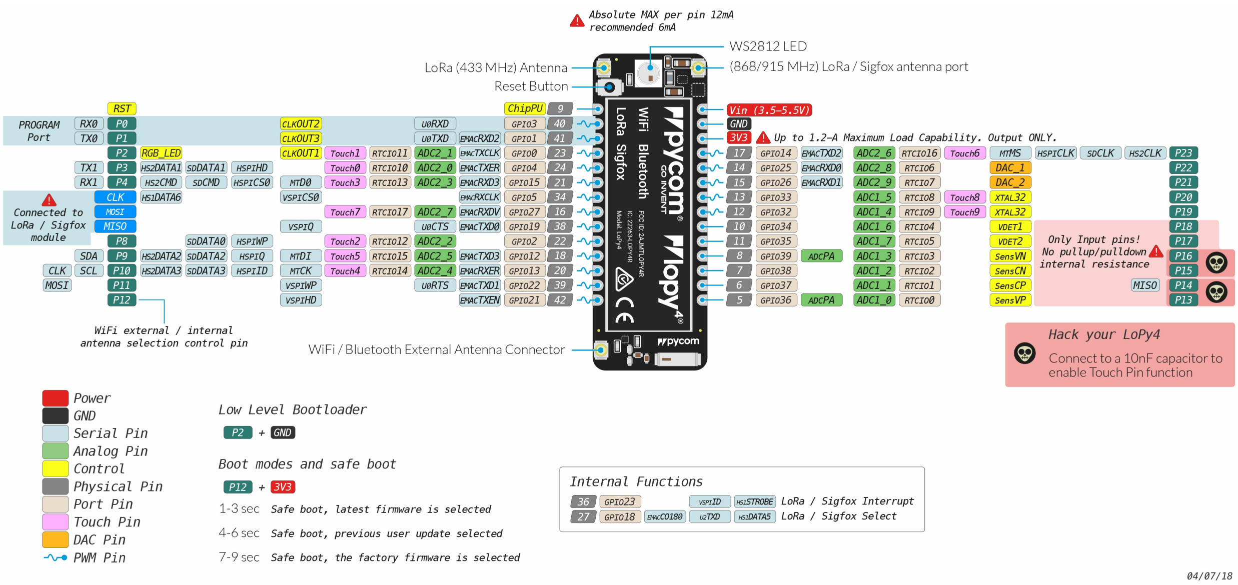


Store: Buy Here

Getting Started Click Here

Pinout

The pinout of the LoPy4 is available as a PDF File.



Please note that the PIN assignments for UART1 (TX1/RX1), SPI (CLK,MOSI,MISO) and I2C (SDA,SCL) are defaults and can be changed in Software.

Datasheet

The datasheet of the LoPy4 is available as a PDF File.

Notes

WiFi

By default, upon boot the LoPy4 will create a WiFi access point with the SSID `lopy4-wlan-xxxx` , where `xxxx` is a random 4-digit number, and the password `www.pycom.io` .

The RF switch that selects between the on-board and external antenna is connected to `P12` , for this reason using `P12` should be avoided unless WiFi is disabled in your application.

Power

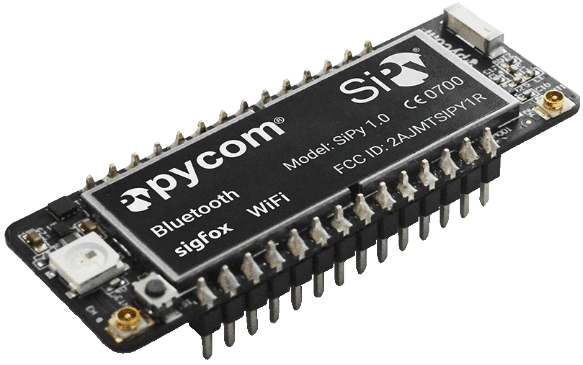
The `Vin` pin on the LoPy4 can be supplied with a voltage ranging from `3.5v` to `5.5v` . The `3.3v` pin on the other hand is output **only**, and must not be used to feed power into the LoPy4, otherwise the on-board regulator will be damaged.

Tutorials

Tutorials on how to the LoPy4 module can be found in the examples section of this documentation. The following tutorials might be of specific interest for the LoPy4:

- WiFi connection
- LoRaWAN node
- LoRaWAN nano gateway
- Sigfox
- BLE

SiPy

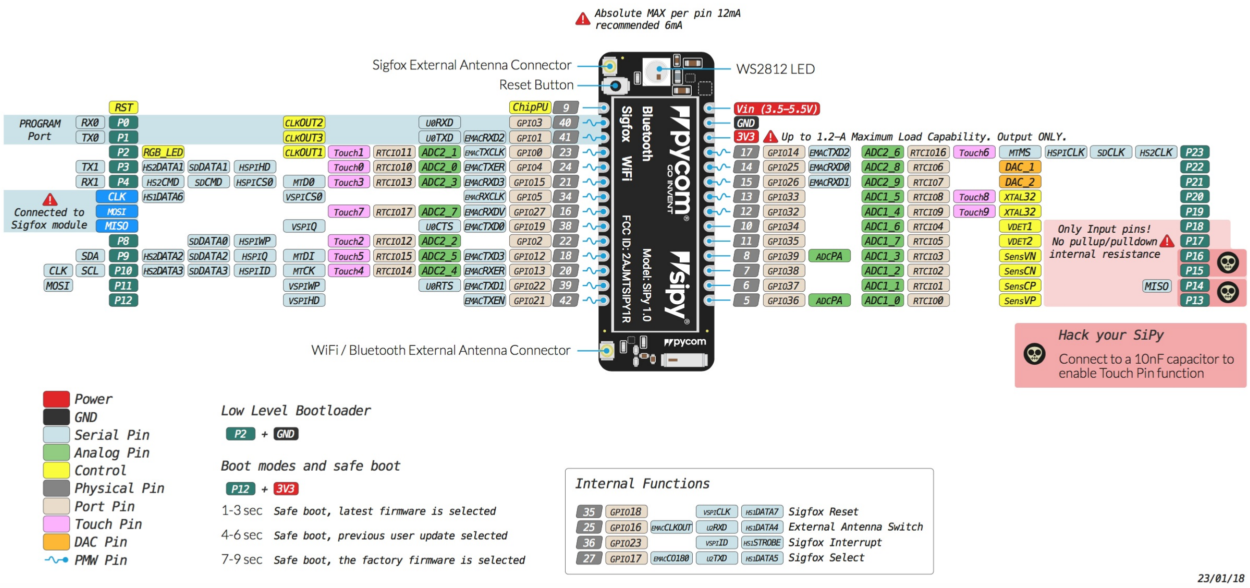


Store: Buy Here

Getting Started Click Here

Pinout

The pinout of the SiPy is available as a PDF File.



Please note that the PIN assignments for UART1 (TX1/RX1), SPI (CLK,MOSI,MISO) and I2C (SDA,SCL) are defaults and can be changed in Software.

Datasheet

The datasheet of the SiPy is available as a PDF File.

Notes

WiFi

By default, upon boot the SiPy will create a WiFi access point with the SSID `sipy-wlan-XXXX`, where `XXXX` is a random 4-digit number, and the password `www.pycom.io`.

Power

The `Vin` pin on the SiPy can be supplied with a voltage ranging from `3.5v` to `5.5v`. The `3.3v` pin on the other hand is output **only**, and must not be used to feed power into the SiPy, otherwise the on-board regulator will be damaged.

Deep Sleep

Due to a couple issues with the SiPy design the module draws more current than it should while in deep sleep. The DC-DC switching regulator always stays in high performance mode which is used to provide the lowest possible output ripple when the modules is in use. In this mode, it draws a quiescent current of 10mA. When the regulator is put into ECO mode, the quiescent current goes down to 10uA. Unfortunately, the pin used to control this mode is out of the RTC domain, and therefore not usable during deep sleep. This causes the regulator to always stay in PWM mode, keeping its quiescent current at 10mA. Alongside this the flash chip doesn't enter power down mode because the CS pin is floating during deep sleep. This causes the flash chip to consume around 2mA of current. To work around this issue a "deep sleep shield" is available that attaches to the module and allows power to be cut off from the device. The device can then be re-enabled either on a timer or via pin interrupt. With the deep sleep shield the current consumption during deep sleep is between 7uA and 10uA depending on the wake sources configured.

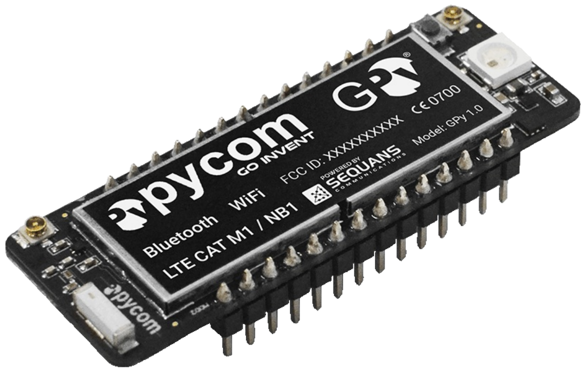
Tutorials

Tutorials on how to the SiPy module can be found in the examples section of this documentation. The following tutorials might be of specific interest for the SiPy:

- WiFi connection
- Sigfox

- BLE

GPy

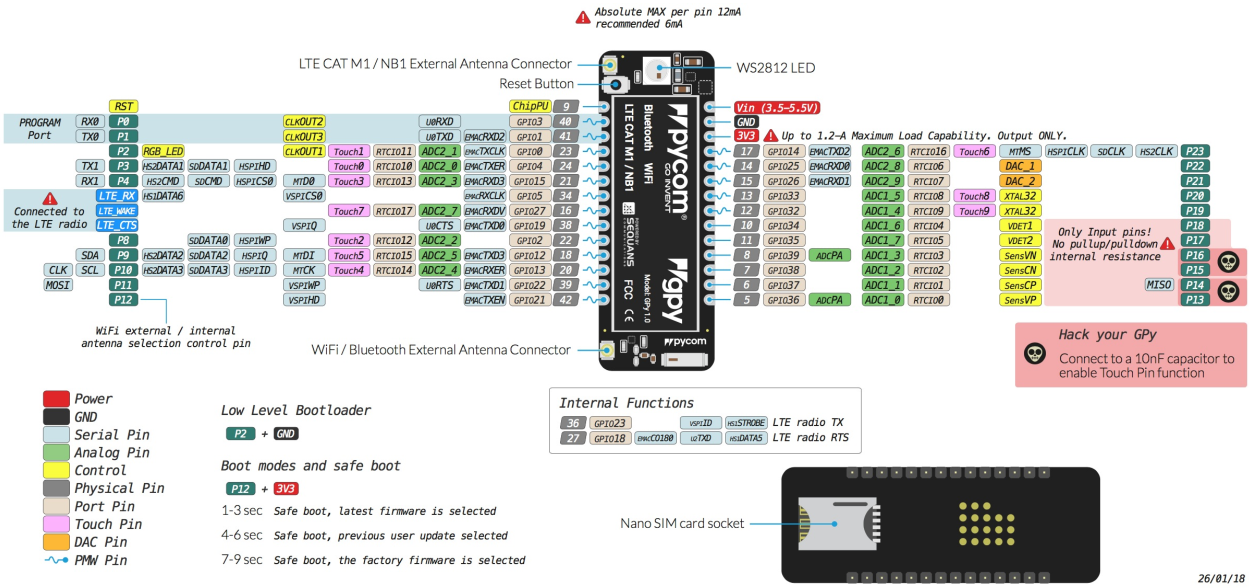


Store: Buy Here

Getting Started Click Here

Pinout

The pinout of the GPy is available as a PDF File.



Please note that the PIN assignments for UART1 (TX1/RX1), SPI (CLK,MOSI,MISO) and I2C (SDA,SCL) are defaults and can be changed in Software.

Datasheet

The datasheet of the GPy is available as a PDF File.

Notes

WiFi

By default, upon boot the GPy will create a WiFi access point with the SSID `gpy-wlan-XXXX` , where `XXXX` is a random 4-digit number, and the password `www.pycom.io` .

The RF switch that selects between the on-board and external antenna is connected to `P12` , for this reason using `P12` should be avoided unless WiFi is disabled in your application.

Power

The `Vin` pin on the GPy can be supplied with a voltage ranging from `3.5v` to `5.5v` . The `3.3v` pin on the other hand is output **only**, and must not be used to feed power into the GPy, otherwise the on-board regulator will be damaged.

AT Commands

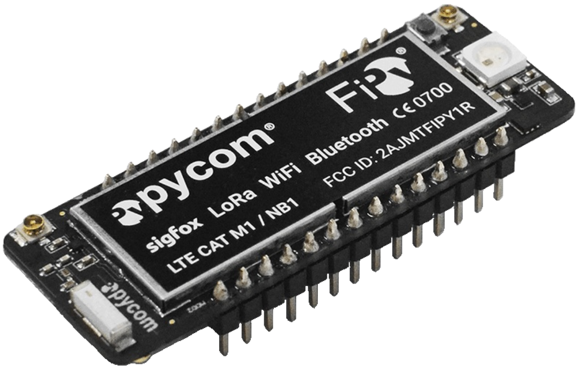
The AT commands for the Sequans Monarch modem on the GPy are available in a PDF file.

Tutorials

Tutorials on how to the GPy module can be found in the examples section of this documentation. The following tutorials might be of specific interest for the GPy:

- WiFi connection
- LTE CAT-M1
- NB-IoT
- BLE

FiPy

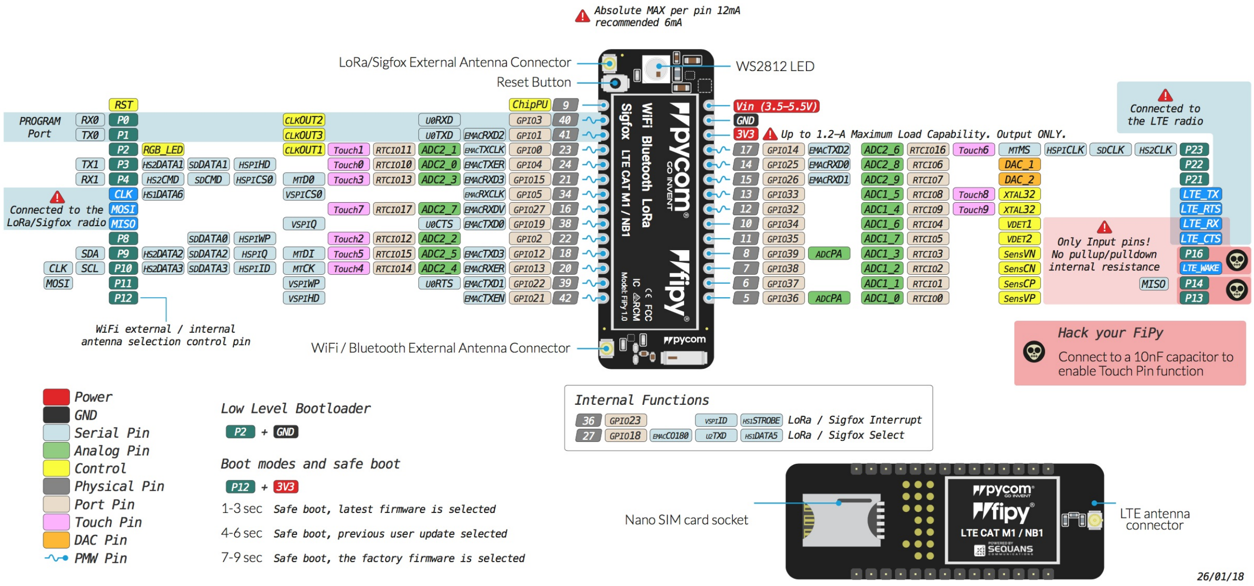


Store: Buy Here

Getting Started Click Here

Pinout

The pinout of the FiPy is available as a PDF File.



Please note that the PIN assignments for UART1 (TX1/RX1), SPI (CLK,MOSI,MISO) and I2C (SDA,SCL) are defaults and can be changed in Software.

Datasheet

The datasheet of the FiPy is available as a PDF File.

Notes

WiFi

By default, upon boot the FiPy will create a WiFi access point with the SSID `fipy-wlan-XXXX` , where `XXXX` is a random 4-digit number, and the password `www.pycom.io` .

The RF switch that selects between the on-board and external antenna is connected to `P12` , for this reason using `P12` should be avoided unless WiFi is disabled in your application.

Power

The `Vin` pin on the FiPy can be supplied with a voltage ranging from `3.5v` to `5.5v` . The `3.3v` pin on the other hand is output **only**, and must not be used to feed power into the FiPy, otherwise the on-board regulator will be damaged.

AT Commands

The AT commands for the Sequans Monarch modem on the FiPy are available in a PDF file.

Tutorials

Tutorials on how to the FiPy module can be found in the examples section of this documentation. The following tutorials might be of specific interest for the FiPy:

- WiFi connection
- LoRaWAN node
- LoRaWAN nano gateway
- Sigfox
- LTE CAT-M1
- NB-IoT
- BLE

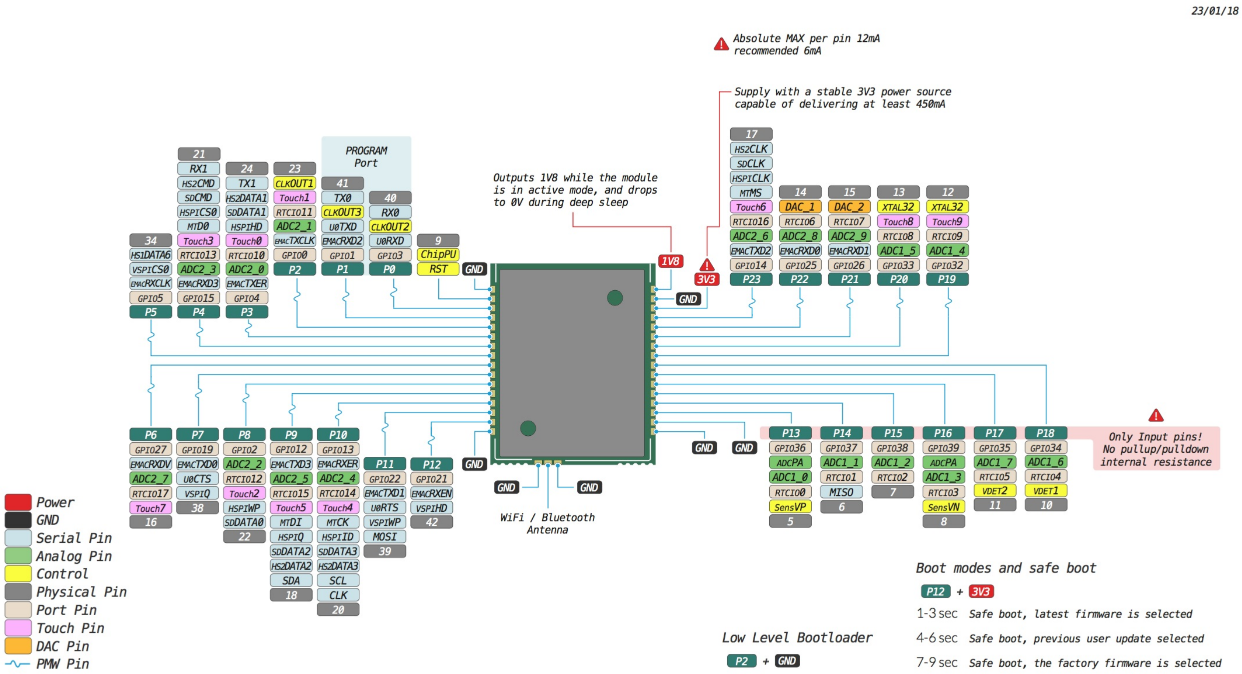
OEM Devices

This section contains all of the datasheets for the Pycom OEM Devices. This includes the W01, L01, L04, and G01.

W01

Pinout

The pinout of the W01 is available as a PDF File.



Specsheets

The specsheet of the W01 is available as a PDF File.

Drawings

The drawings for the W01 is available as a PDF File.

Please note that the PIN assignments for UART1 (TX1/RX1), SPI (CLK, MOSI, MISO) and I2C (SDA, SCL) are defaults and can be changed in Software.

Tutorials

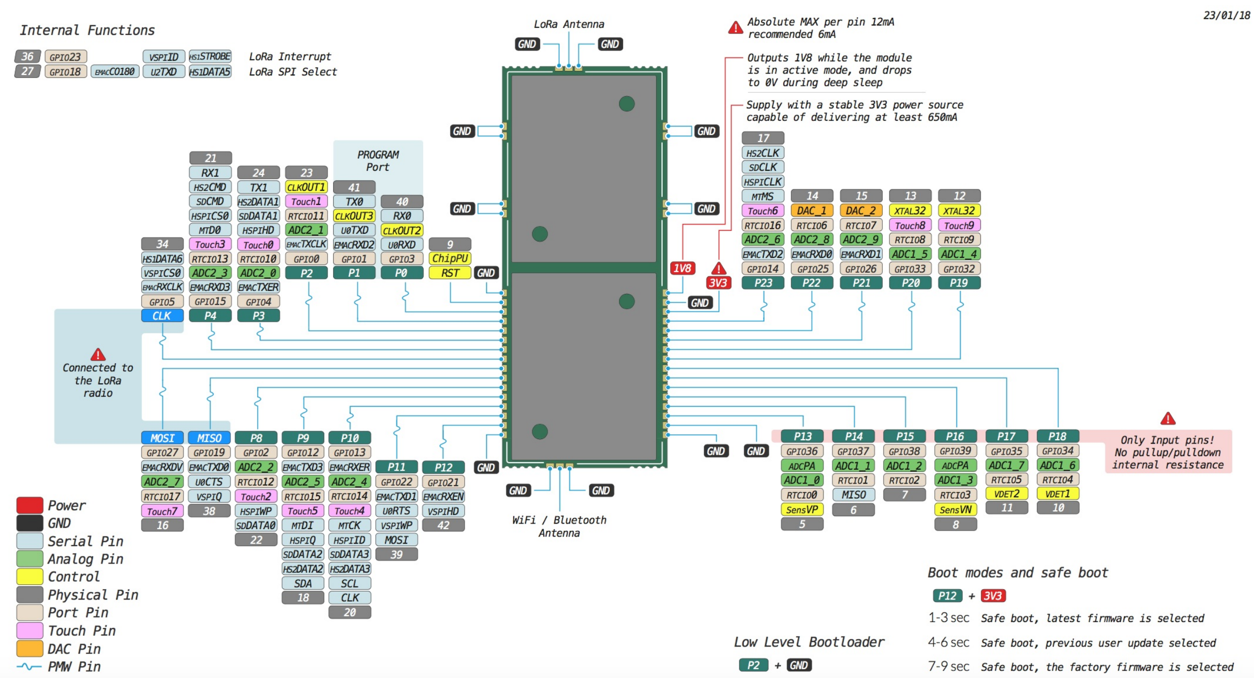
Tutorials on how to the W01 module can be found in the examples section of this documentation. The following tutorials might be of specific interest for the W01:

- WiFi connection
- BLE

L01

Pinout

The pinout of the L01 is available as a PDF File.



Specsheets

The specsheet of the L01 is available as a PDF File.

Drawings

The drawings for the L01 is available as a PDF File.

Please note that the PIN assignments for UART1 (TX1/RX1), SPI (CLK, MOSI, MISO) and I2C (SDA, SCL) are defaults and can be changed in Software.

Tutorials

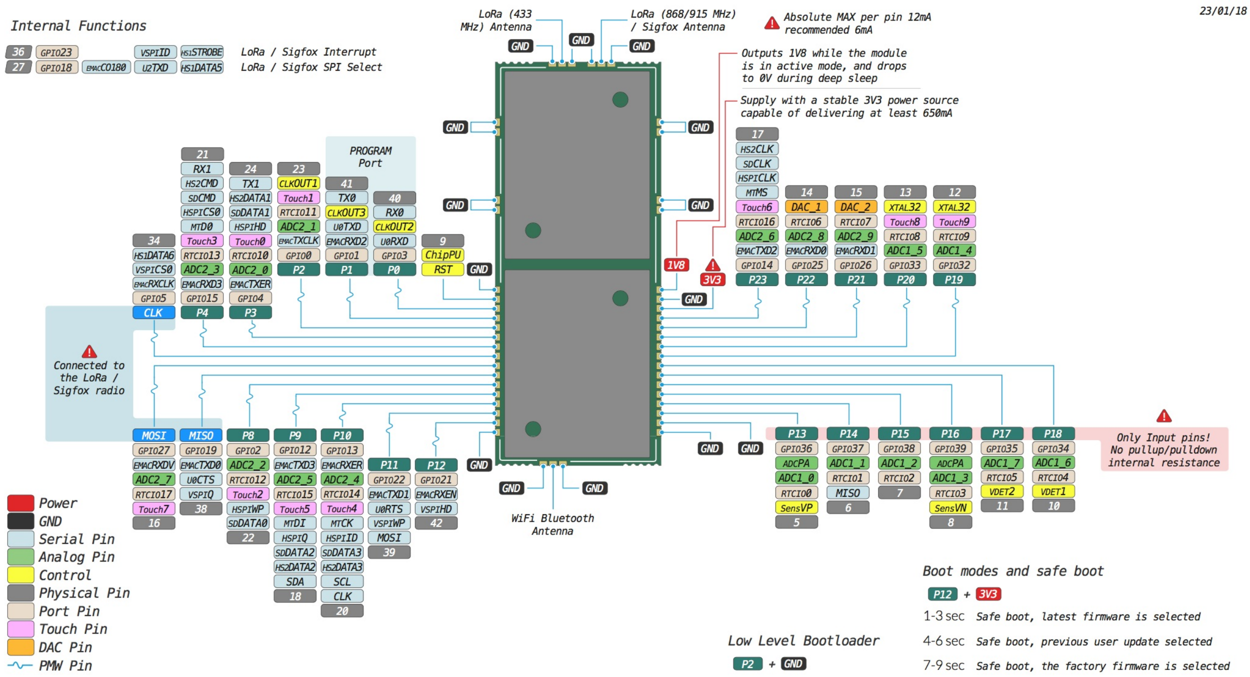
Tutorials on how to the L01 module can be found in the examples section of this documentation. The following tutorials might be of specific interest for the L01:

- WiFi connection
- LoRaWAN node
- LoRaWAN nano gateway
- BLE

L04

Pinout

The pinout of the L04 is available as a PDF File.



Specsheets

The specsheet of the L04 is available as a PDF File.

Drawings

The drawings for the L04 is available as a PDF File.

Please note that the PIN assignments for UART1 (TX1/RX1), SPI (CLK, MOSI, MISO) and I2C (SDA, SCL) are defaults and can be changed in Software.

Tutorials

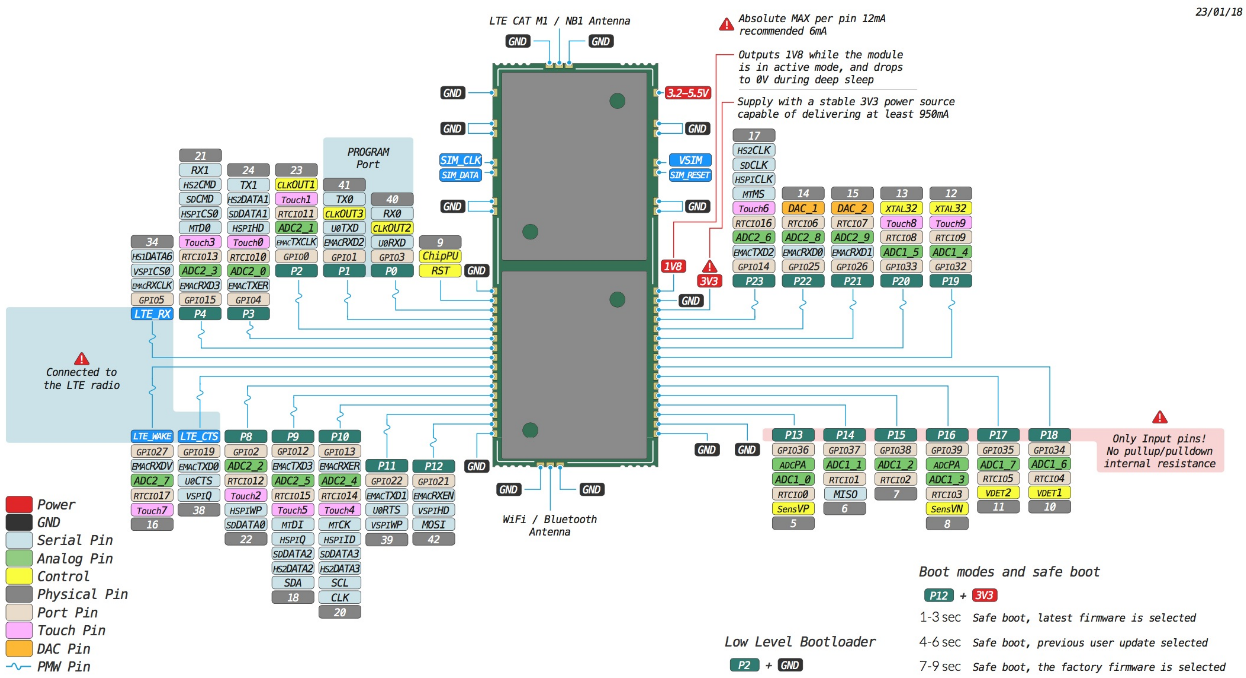
Tutorials on how to the L04 module can be found in the examples section of this documentation. The following tutorials might be of specific interest for the L04:

- WiFi connection
- LoRaWAN node
- LoRaWAN nano gateway
- Sigfox
- BLE

G01

Pinout

The pinout of the G01 is available as a PDF File.



Specsheets

The specsheet of the G01 is available as a PDF File.

Drawings

The drawings for the G01 is available as a PDF File.

Please note that the PIN assignments for UART1 (TX1/RX1), SPI (CLK, MOSI, MISO) and I2C (SDA, SCL) are defaults and can be changed in Software.

AT Commands

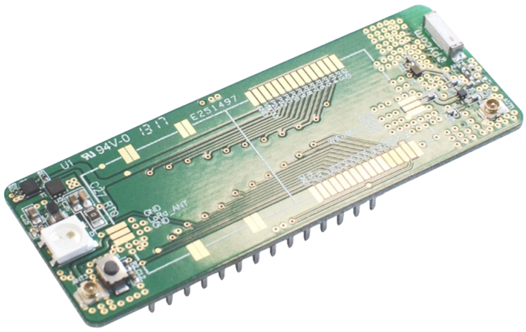
The AT commands for the Sequans Monarch modem on the G01 are available in a PDF file.

Tutorials

Tutorials on how to the G01 module can be found in the examples section of this documentation. The following tutorials might be of specific interest for the G01:

- WiFi connection
- LTE CAT-M1
- NB-IoT
- BLE

L01 reference design



The L01 OEM reference board is a reference design suitable L01 as well as W01 making it possible to have a single PCB design that can accommodate both OEM modules.

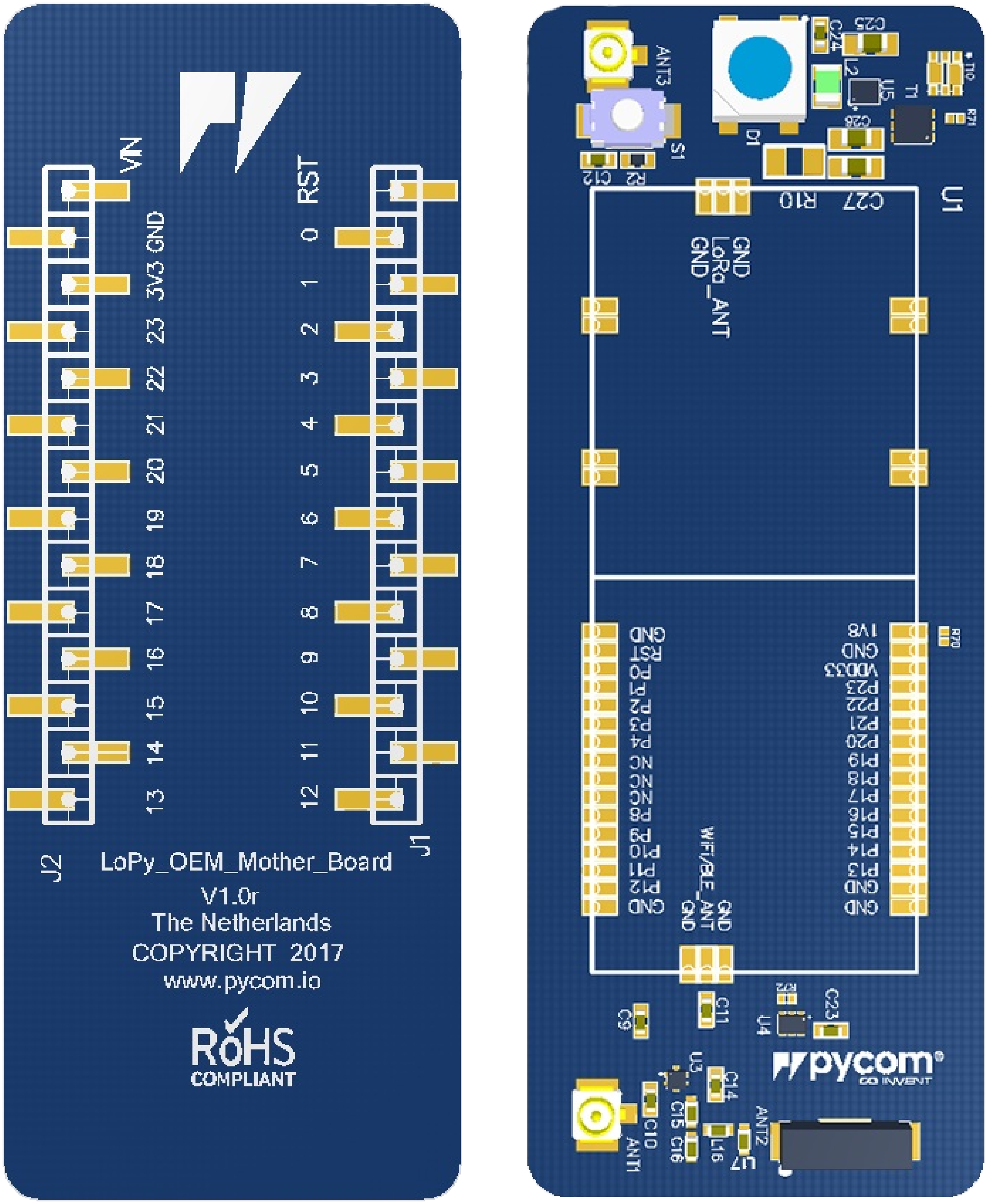
If you require a reference board for the L04 or G01, this design is **not** suitable as it does not feature a SIM slot or the double antenna connection. For the G01 or L04 please use the Universal OEM Baseboard Reference

Features

- Suits both L01 or W01 OEM Modules
- U.FL connector for the L01's LoRa output.
- On-board 2.4GHz antenna for WiFi and Bluetooth, with the ability to switch to a external antenna via a U.FL connector.
- WS2812B RGB LED
- 3.5-5.5 Input switch mode DC-DC regulator with low current draw during deep sleep
- Reset button

Layout

The layout of the L01 baseboard reference is available as a PDF File.



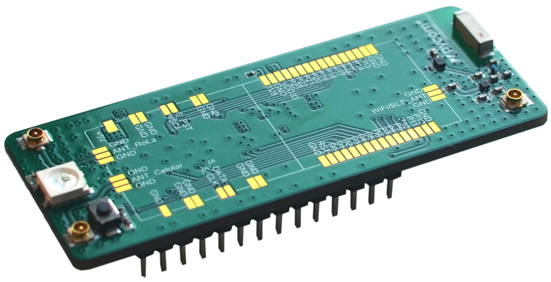
Schematic

The schematic of the L01 baseboard reference is available as a PDF File.

Altium Project and Gerber Files

The Altium Project and Gerber files are also available as a ZIP File.

OEM Baseboard Reference Design Files



The universal OEM reference board is a reference design suitable W01, L01, L04 and G01 OEM modules, making it possible to have a single PCB design that can accommodate all our OEM modules.

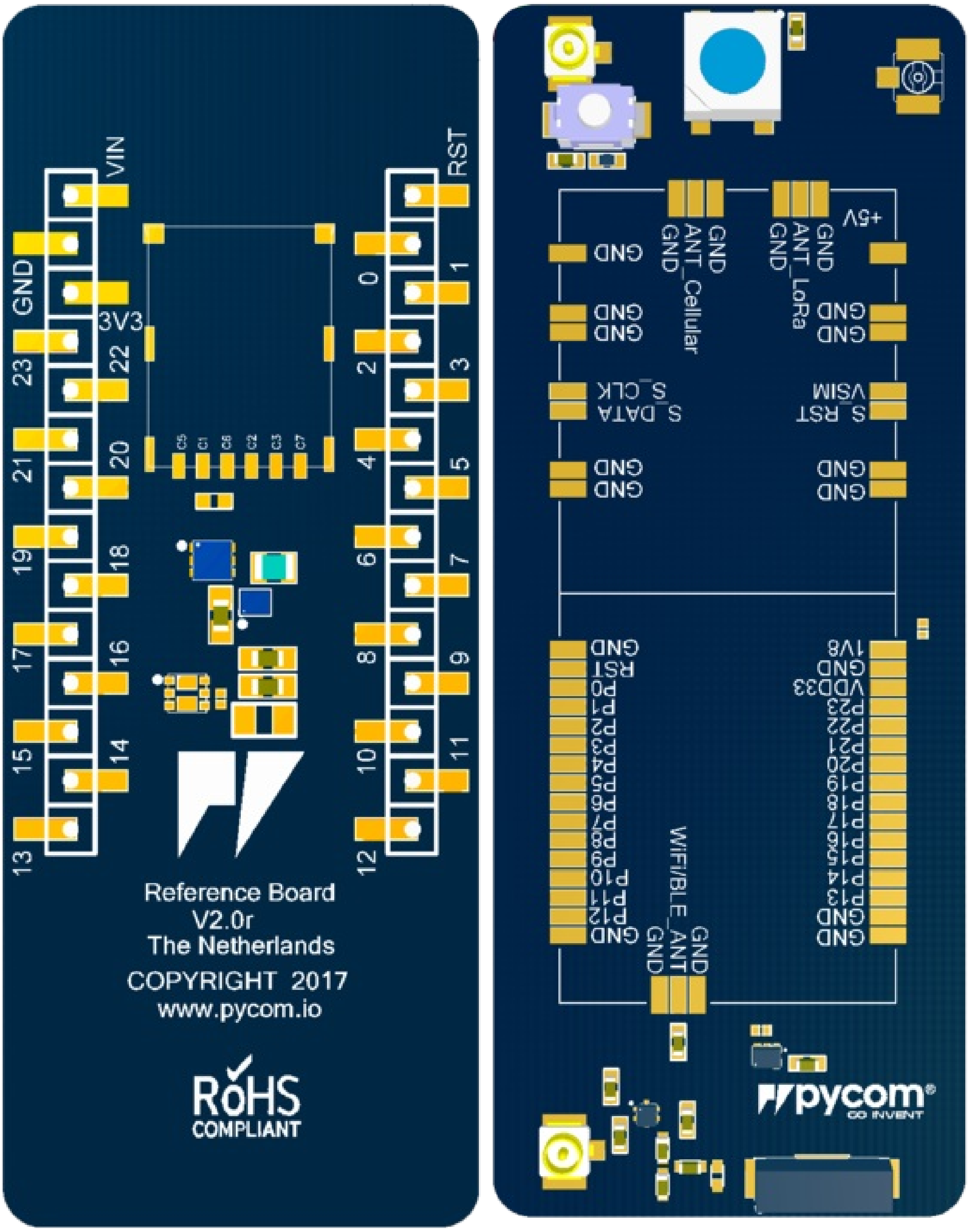
If you require a reference board for the G01, only this design is suitable. The L01 reference board does not contain the necessary SIM slot.

Features

- Suits all OEM modules (L01, L04, W01, G01)
- On-board 2.4GHz antenna for WiFi and Bluetooth, with the ability to switch to a external antenna via a U.FL connector.
- 3 U.FL connectors for all the outputs available on the OEM modules
- WS2812B RGB LED
- 3.5-5.5 Input switch mode DC-DC regulator with low current draw during deep sleep
- Reset button

Layout

The layout of the OEM baseboard reference is available as a PDF File.



Schematic

The schematic of the OEM baseboard reference is available as a PDF File.

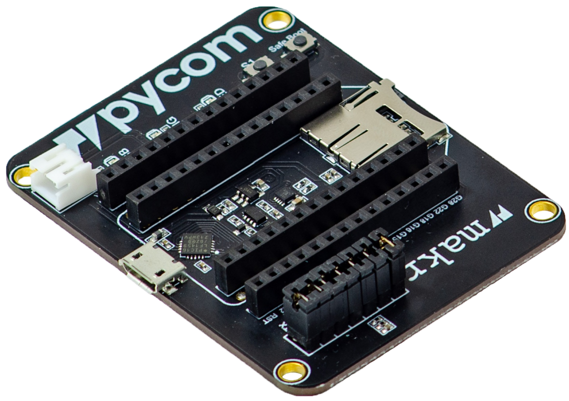
Altium Project and Gerber Files

The Altium Project and Gerber files are also available as a ZIP File.

Expansion Boards and Shields

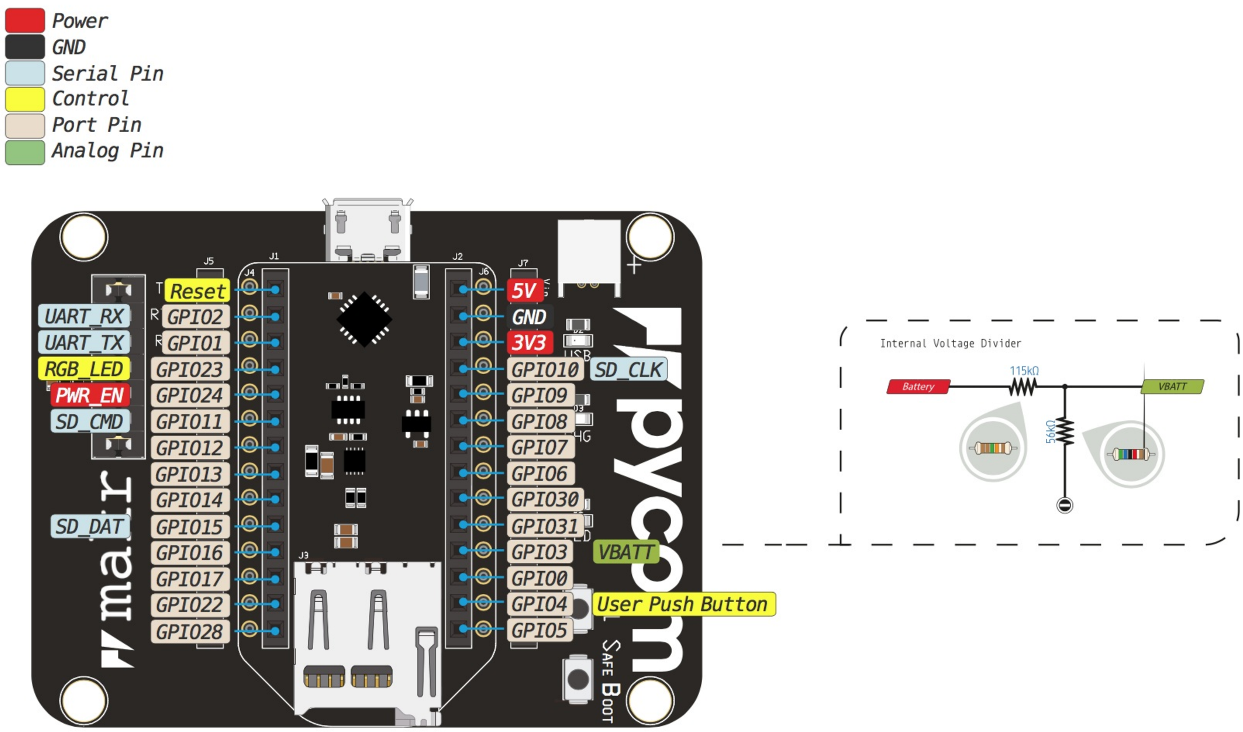
This section contains all of the datasheets for the Pycom Expansion Boards and Shields. This includes the Expansion Board, Pytrack, Pysense and Deep Sleep Shield.

Expansion Board 3.0



Pinout

The pinout of the Expansion Board is available as a PDF File.



Be gentle when plugging/unplugging from the USB connector. Whilst the USB connector is soldered and is relatively strong, if it breaks off it can be very difficult to fix.

Battery Charger

The Expansion Board features a single cell Li-Ion/Li-Po charger. When the board is being powered via the micro USB connector, the Expansion Board will charge the battery (if connected). When the `CHG` jumper is present the battery will be charged at `450mA`. If this value is too high for your application, removing the jumper lowers the charge current to `100mA`.

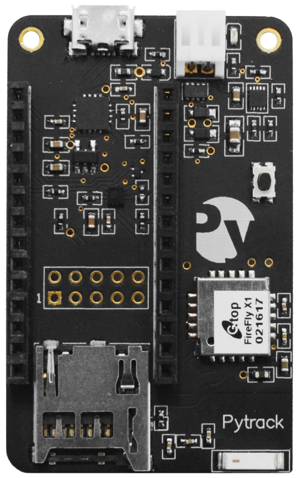
Specsheets

The specsheet of the Expansion Board is available as a PDF File.

Differences between v2.0 and v3.0

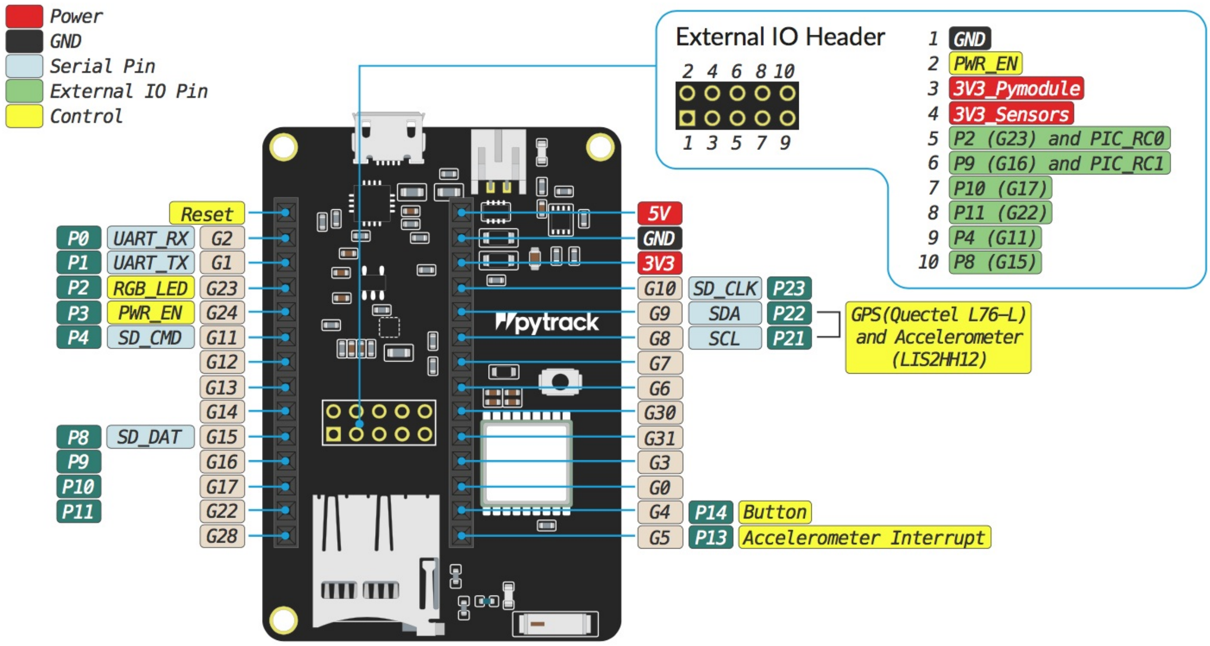
- The FTDI chip as been replaced with a custom programmed PIC like on the Pysense/Pytrack/Pyscan boards. This allows our firmware update tool to automatically put the module into bootloader mode.
- Added a "Safe boot" button to enter safe boot easier. This button connects `P12` to `3.3v` and if pressed and held while the reset button is pressed on a Pycom module, the module will enter safe boot.

Pytrack



Pinout

The pinout of the Pytrack is available as a [PDF File](#).



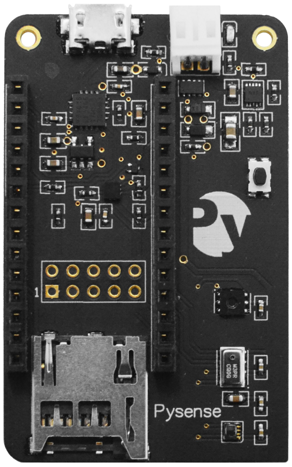
Battery Charger

The board features a single cell Li-Ion/Li-Po charger. When the board is being powered via the micro USB connector, it will charge the battery (if connected).

Specsheets

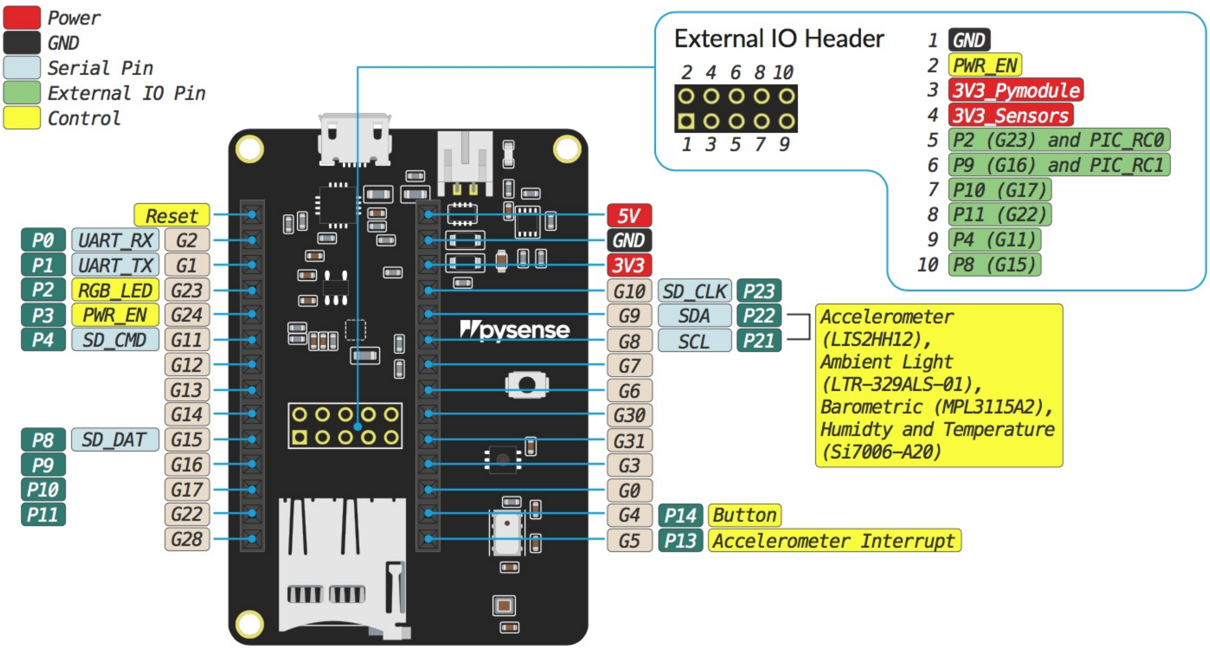
The specsheet of the Pytrack is available as a PDF File.

Pysense



Pinout

The pinout of the Pysense is available as a PDF File.



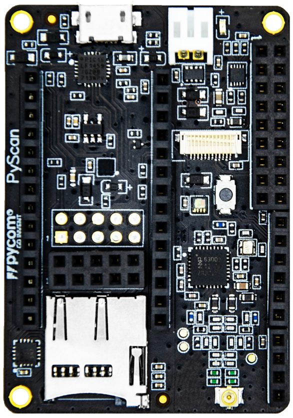
Battery Charger

The board features a single cell Li-Ion/Li-Po charger. When the board is being powered via the micro USB connector, it will charge the battery (if connected).

Specsheets

The specsheet of the Pysense is available as a PDF File.

Pyscan



Pyscan Libraries

Pyscan libraries to use the RFID/NFC reader are located here: https://github.com/pycom/pycom-libraries/tree/master/pyscan The accelerometer library is here: https://github.com/pycom/pycom-libraries/blob/master/pytrack/lib/LIS2HH12.py

For the time being, we recommend to upload the `MFRC630.mpy` file via FTP due to current limitations of Pymakr that will be fixed shortly.

Libraries for the rest of the components will be added soon.

Pyscan components:

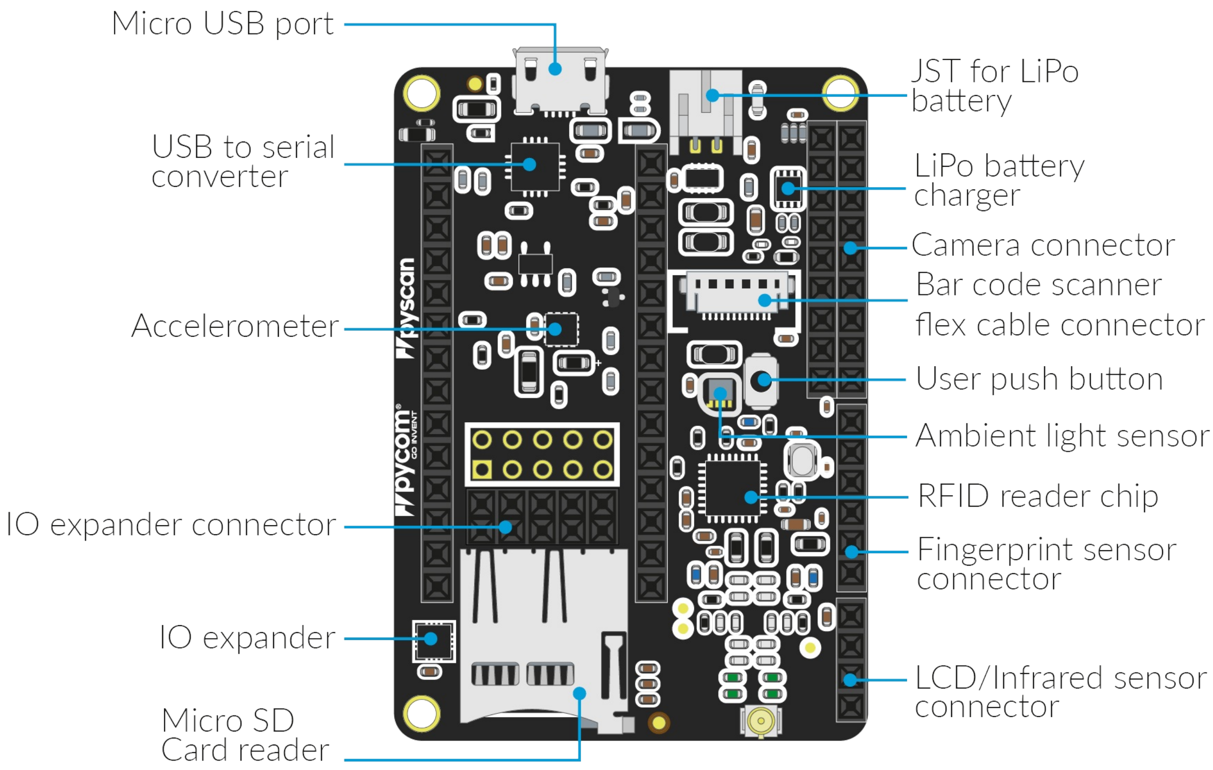
- **Accelerometer**: ST LIS2HH12
- **Ambient light sensor**: Lite-on LTR-329ALS-01
- **RFID/NFC reader**: NXP MFRC63002HN, 151

Driver

The Windows 7 driver for Pyscan is located in: https://docs.pycom.io/chapter/pytrackpysense/installation/drivers.html For other Operating Systems there's no driver required.

Pinout

The pinout of the Pyscan is available as a PDF File.



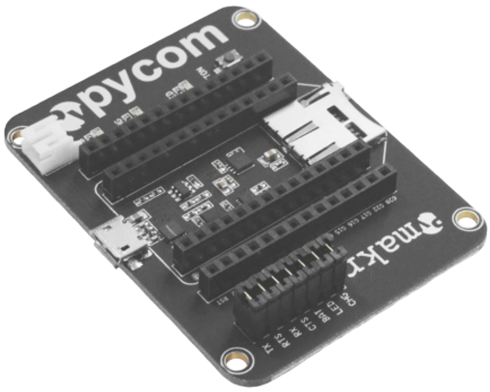
Battery Charger

The board features a single cell Li-Ion/Li-Po charger. When the board is being powered via the micro USB connector, it will charge the battery (if connected).

Specsheets

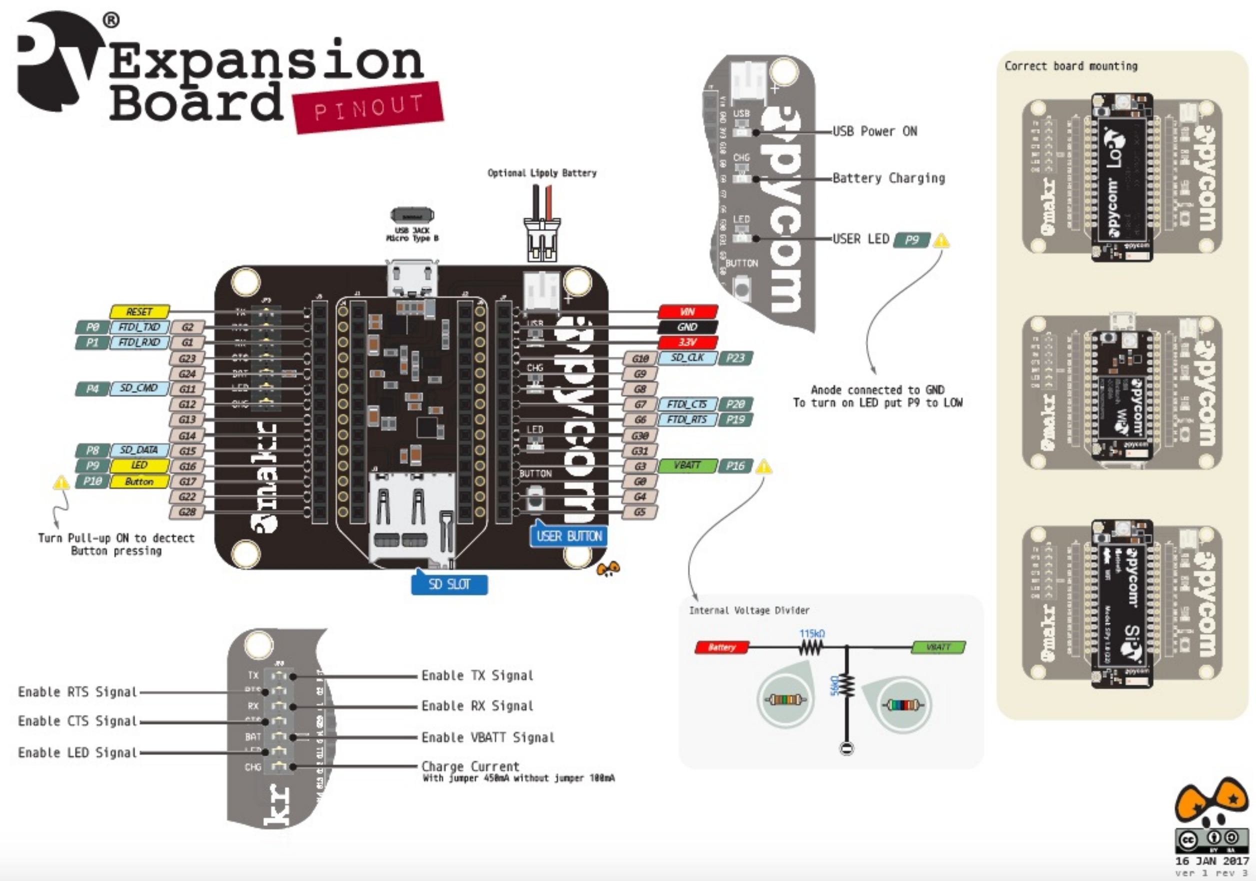
The specsheet of the Pyscan is available as a PDF File.

Expansion Board 2.0



Pinout

The pinout of the Expansion Board is available as a PDF File.



Be gentle when plugging/unplugging from the USB connector. Whilst the USB connector is soldered and is relatively strong, if it breaks off it can be very difficult to fix.

Battery Charger

The Expansion Board features a single cell Li-Ion/Li-Po charger. When the board is being powered via the micro USB connector, the Expansion Board will charge the battery (if connected). When the `CHG` jumper is present the battery will be charged at `450mA`. If this value is too high for your application, removing the jumper lowers the charge current to `100mA`.

Specsheets

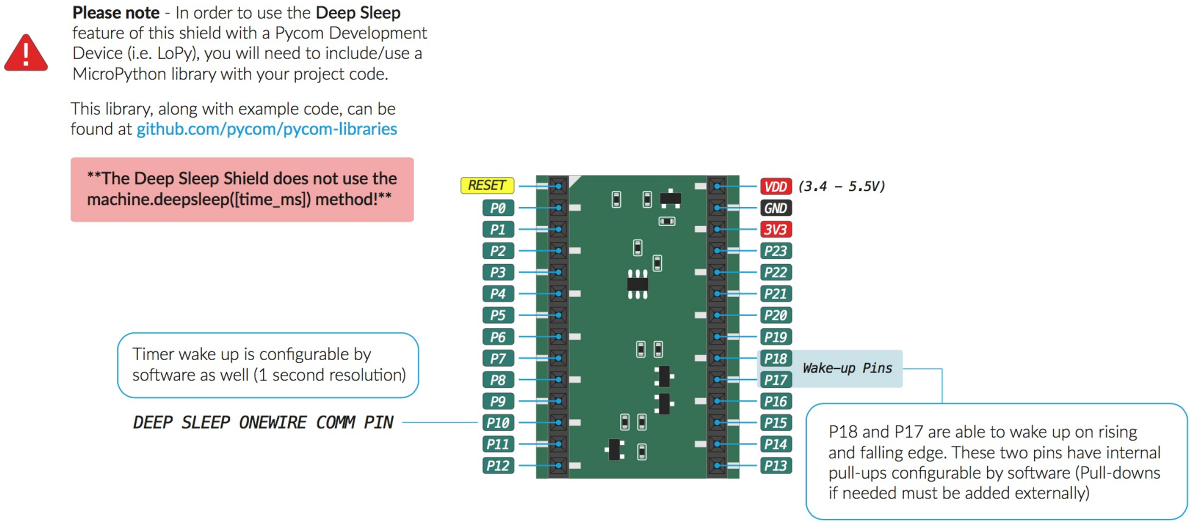
The specsheet of the Expansion Board is available as a PDF File.

Deep Sleep Shield

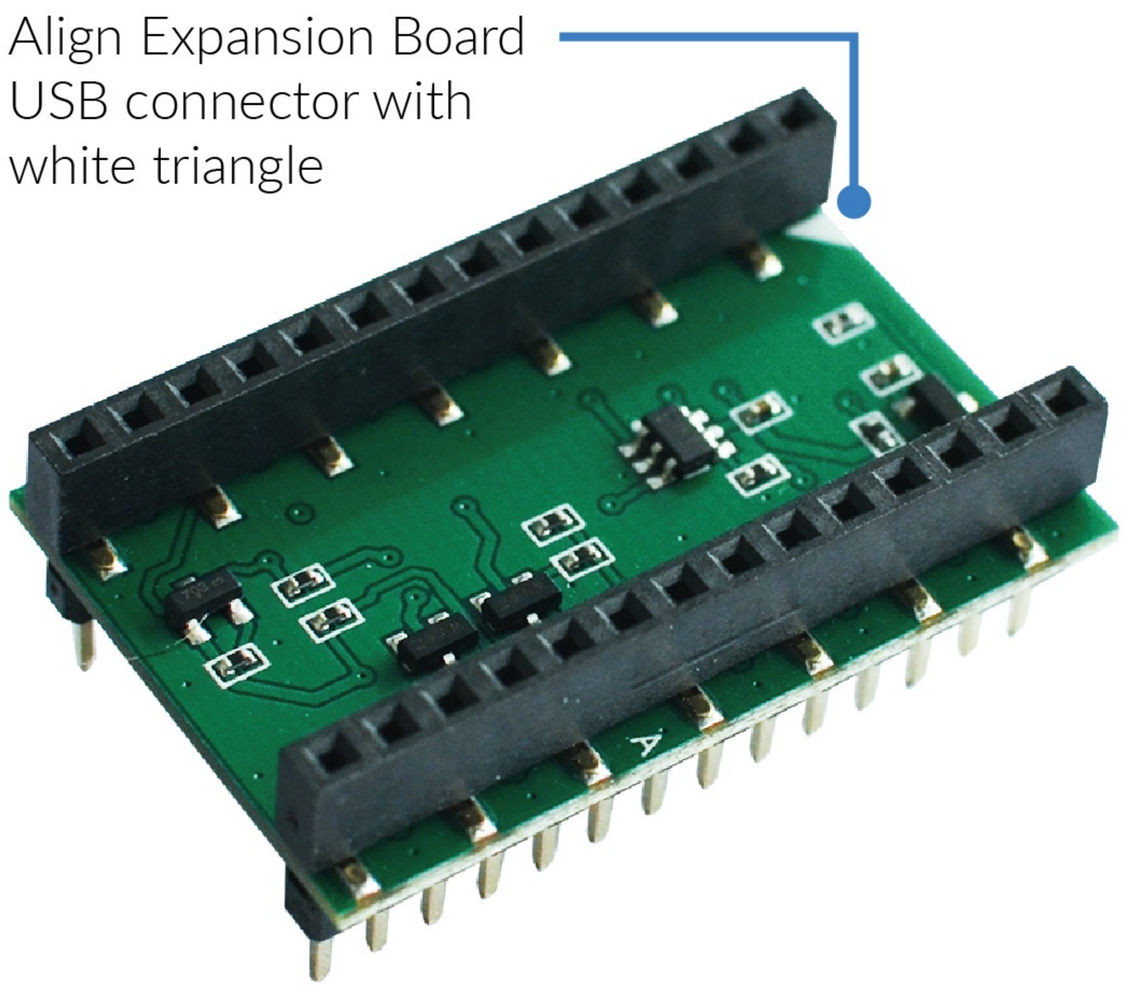
The schematic of the Deep Sleep Shield is available as a PDF File.

Pinout

The pinout of the Deep Sleep Shield is available as a PDF File.



To correctly connect a WiPy 2.0, LoPy or SiPy to the Deep Sleep Shield, align the white triangle on the Shield with the LED of the Pycom Device. Once the Pycom Device is seated onto the Deep Sleep Shield, this can then be connected to the Expansion Board.


Align Expansion Board
USB connector with
white triangle
A

Deep Sleep API

This chapter describes the library which controls the Deep Sleep Shield. This includes the controls for external interrupts and timer setup of the deep sleep functionality.

To use this library, please upload the associated Deep Sleep Library to `/lib` on the target Pycom device.

Quick Example

```
from deepsleep import DeepSleep
import deepsleep

ds = DeepSleep()

# get the wake reason and the value of the pins during wake up
wake_s = ds.get_wake_status()
print(wake_s)

if wake_s['wake'] == deepsleep.PIN_WAKE:
    print("Pin wake up")
elif wake_s['wake'] == deepsleep.TIMER_WAKE:
    print("Timer wake up")
else:  # deepsleep.POWER_ON_WAKE:
    print("Power ON reset")

ds.enable_pullups('P17')  # can also do ds.enable_pullups(['P17', 'P18'])
ds.enable_wake_on_fall('P17') # can also do ds.enable_wake_on_fall(['P17', 'P18'])

ds.go_to_sleep(60)  # go to sleep for 60 seconds
```

DeepSleep

The Deep Sleep Shield allows for waking up via a user trigger and also via an external interrupt (i.e. Accelerometer, Button).

Constructors

class **DeepSleep()**

Creates a DeepSleep object, that will control the board's sleep features. For example;

```
ds = DeepSleep()
```

Methods

deepsleep.enable_auto_poweroff()

This method allows for a critical battery voltage to be set. For example, if the external power source (e.g. LiPo Cell) falls below `3.3V`, turn off the Pycom device. This is intended to protect the hardware from under voltage.

deepsleep.enable_pullups(pins)

This method allows for pull-up pins to be enabled. For example, if an external trigger occurs, wake the Pycom device from Deep Sleep. `pins` may be passed into the method as a list, i.e. `['P17', 'P18']`.

deepsleep.disable_pullups(pins)

This method allows for pull-up pins to be disabled. For example, if an external trigger occurs, wake the Pycom device from Deep Sleep. `pins` may be passed into the method as a list, i.e. `['P17', 'P18']`.

deepsleep.enable_wake_on_raise(pins)

This method allows for pull-up pins to trigger on a rising voltage. For example, if an external rising voltage triggers occurs, wake the Pycom device from Deep Sleep. `pins` may be passed into the method as a list, i.e. `['P17', 'P18']`.

deepsleep.disable_wake_on_raise(pins)

This method allows for disabling pull-up pins that trigger on a rising voltage. `pins` may be passed into the method as a list, i.e. `['P17', 'P18']`.

deepsleep.enable_wake_on_fall(pins)

This method allows for pull-up pins to trigger on a falling voltage. For example, if an external falling voltage triggers occurs, wake the Pycom device from Deep Sleep. `pins` may be passed into the method as a list, i.e. `['P17', 'P18']`.

deepsleep.disable_wake_on_fall(pins)

This method allows for disabling pull-up pins that trigger on a falling voltage. `pins` may be passed into the method as a list, i.e. `['P17', 'P18']`.

deepsleep.get_wake_status()

This method returns the status of the pins at wakeup from deep sleep. The method returns a `dict` with the states of `wake`, `P10`, `P17`, `P18`.

deepsleep.set_min_voltage_limit(value)

This method relates to the `enable_auto_poweroff` method and allows the user to specify the minimum power off voltage as a value.

deepsleep.go_to_sleep(seconds)

This method sends the board into deep sleep for a period of `seconds` or until an external interrupt has triggered (see `set_pullups`).

deepsleep.hw_reset()

This method resets the PIC controller and resets it to the state previous to the pins/min-voltage being set.

> Please note that more functionality is being added weekly to these libraries. If a required feature is not available, feel free to contribute with a pull request at the Pycom Libraries GitHub repository.

Notes

Powering with an external power source

The devices can be powered by a battery or other external power source.

Be sure to connect the positive lead of the power supply to `VIN`, and ground to `GND`.

When powering via `VIN`:

- The input voltage must be between `3.4V` and `5.5V`.

Please **DO NOT** power the board via the `3.3V` pin as this may damage the device. ONLY use the `VIN` pin for powering Pycom devices.

The battery connector for the Expansion Board is a **JST PHR-2** variant. The Expansion Board exposes the male connector and an external battery should use a female adapter in order to connect and power the expansion board. The polarity of the battery should be checked before being plugged into the expansion board, the cables may require swapping.

The `GPIO` pins of the modules are **NOT** `5V` tolerant, connecting them to voltages higher than `3.3V` might cause irreparable damage to the device.

Static electricity can damage components on the device and may destroy them. If there is a lot of static electricity in the area (e.g. dry and cold climates), take extra care not to shock the device. If the device came in a ESD bag (Silver packaging), the best way to store and carry the device is inside this bag as it will be protected against static discharges.

Development Modules Datasheets

OEM Module Datasheets

Expansion Board and Shield Datasheets

What is Pybytes?

Pybytes is an IoT Ecosystem that empowers you by granting full control of all your Pycom devices.

With Pybytes you have control over your device's data stream and more:

- Visualise sensors data according to your interests using our customisable dashboard;
- Check the status of your entire fleet;
- Keep track of your assets with our geolocation feature;
- Distribute firmware updates on a scalable approach.

In a nutshell, Pybytes is an environment designed to optimise your IoT applications using Pycom boards.

What Pybytes offers you?

- Data Visualisation: Pybytes dashboard is customisable, allowing you to freely set up key performance indicators and time series data from all your sensors.
- Intelligent notifications: Keep track of your device's status, battery level, data streaming and measurements with pre-defined alarms. Receive notifications via email or SMS.
- Terminal: Execute commands to gather accurate information from your devices using Pybytes terminal shell.
- Firmware updates over the air: Upgrade or downgrade firmware versions with our exclusive firmware update.
- Track your assets position: Google Maps API empowers your view over your device's geolocation.

Let's get started!

Getting started with Pybytes

Connect your Pycom module to Pybytes

Visualise data from your device

Create your Pybytes account

Follow these steps to create a Pybytes account:

Step 1: Go to the registration page

1. Go to this link.
2. Enter your full name, email address and a password to your account.
3. Confirm the verification message sent to your email address.
4. Click on the link and complete your login.

Go Invent!

Now it's time to explore Pybytes. You can start by connecting your Pycom board to Pybytes. Check here!

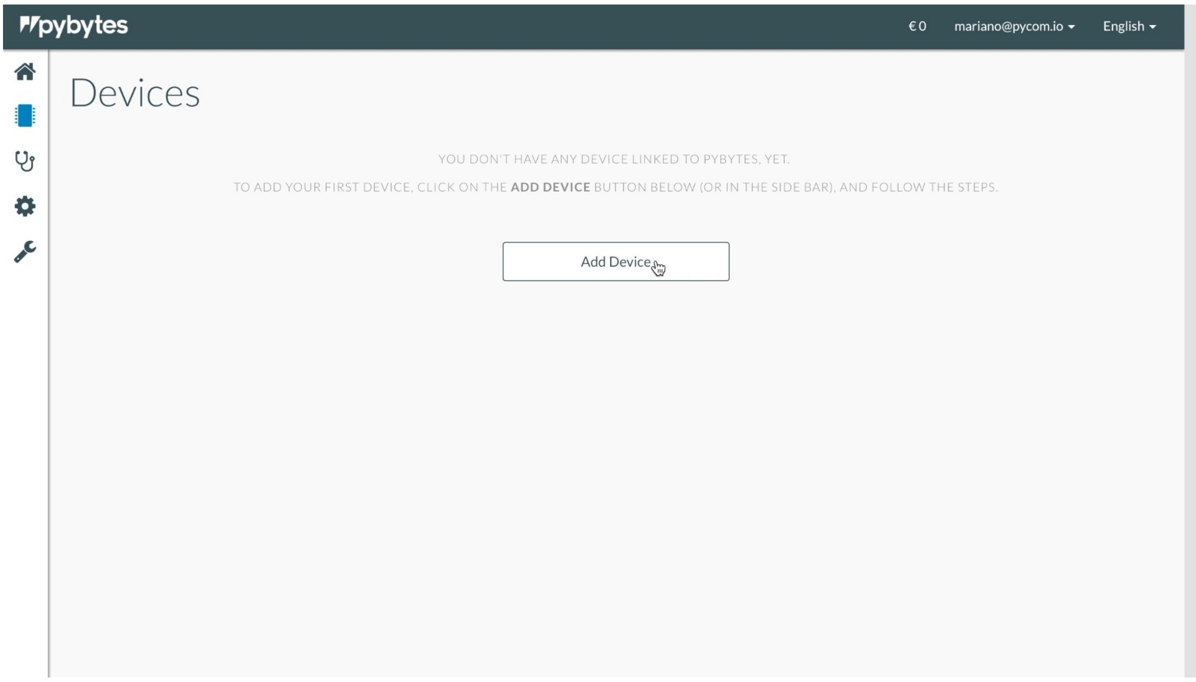
Add a device to pybytes

In this section, we will explain to you how to add a device to Pybytes

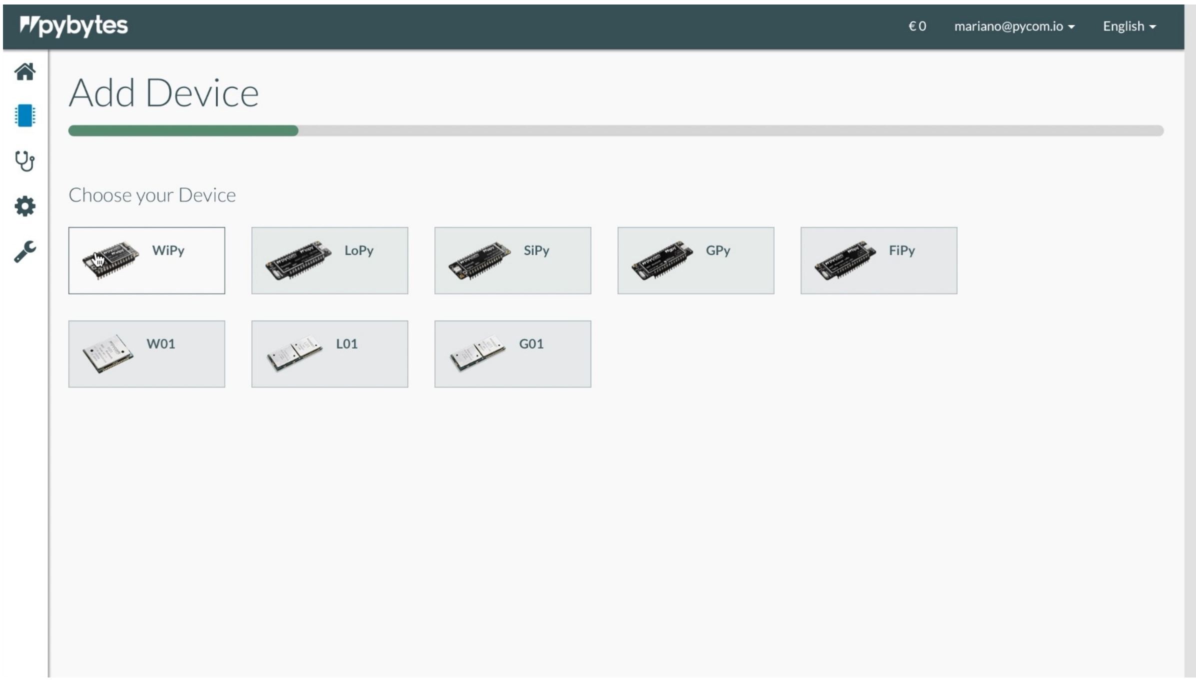
Step 1: Add device wizard

In Pybytes, go to `Devices` Page:

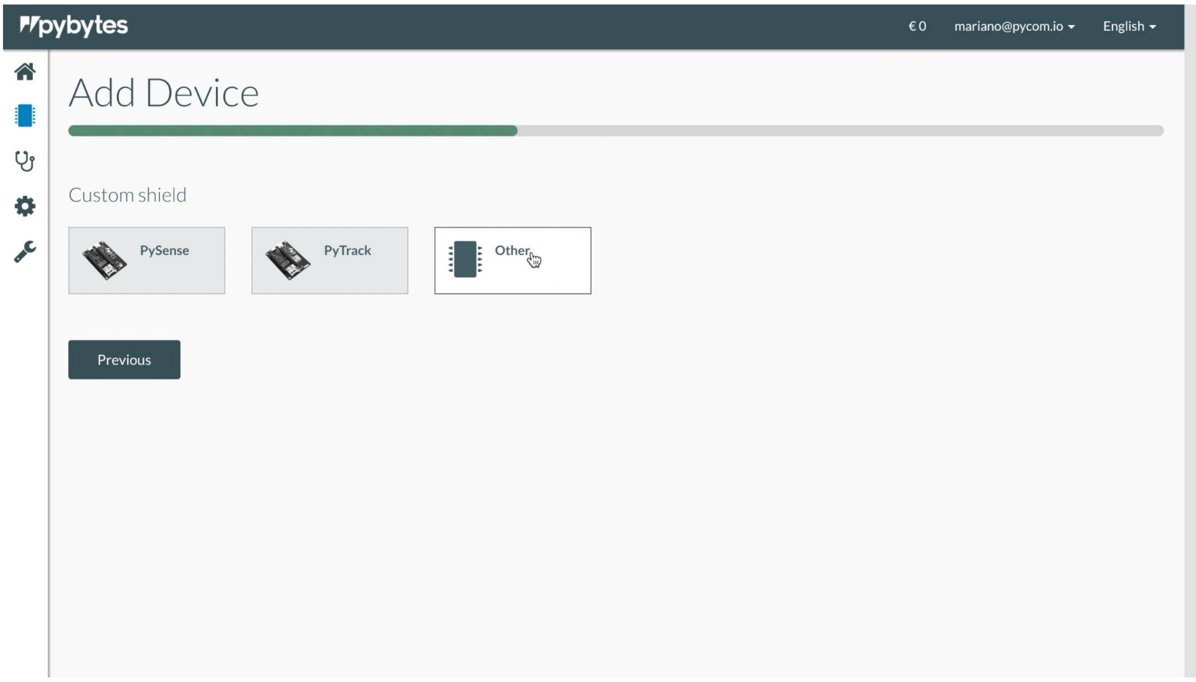
1. Click on `Add Device` .



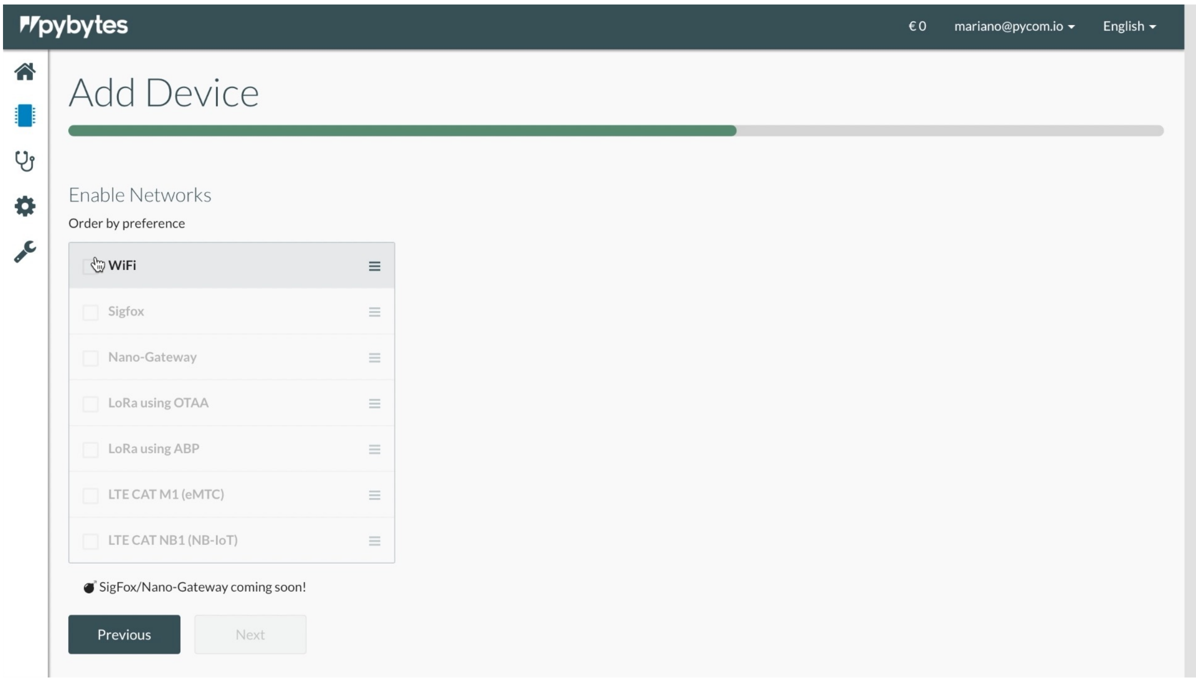
2. Select your device (e.g., WiPy, LoPy, SiPy, etc.);



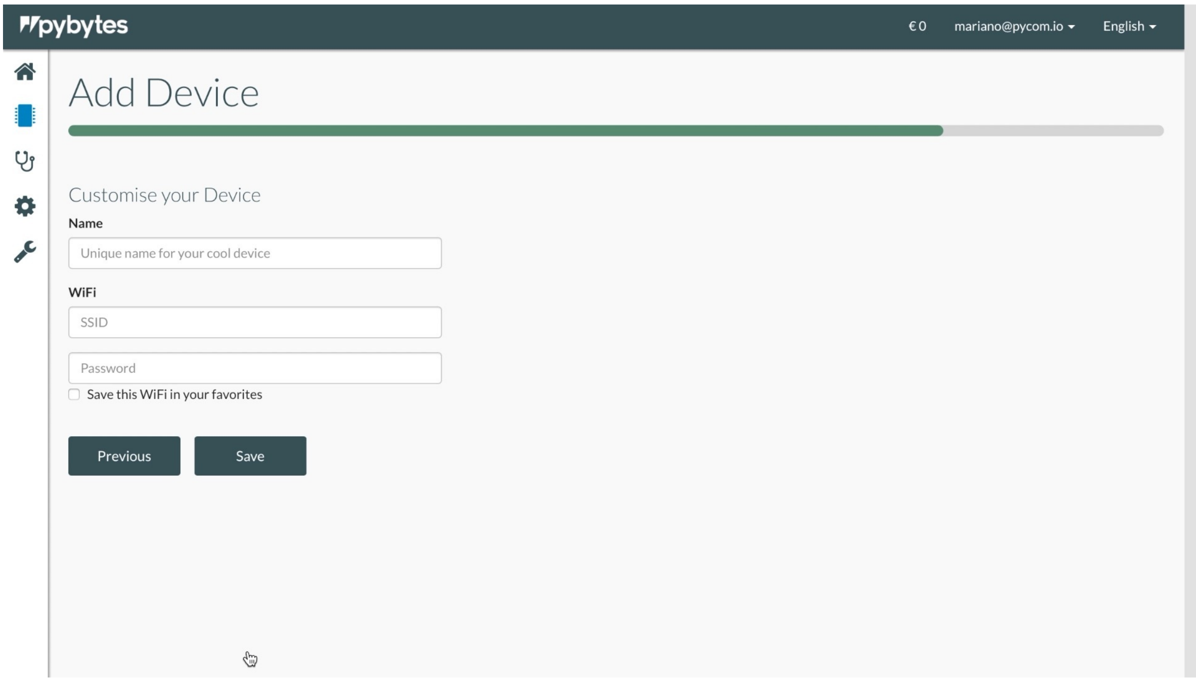
3. Select your shield (e.g., PySense, PyTrack, PyScan or other);



4. Select your network option;

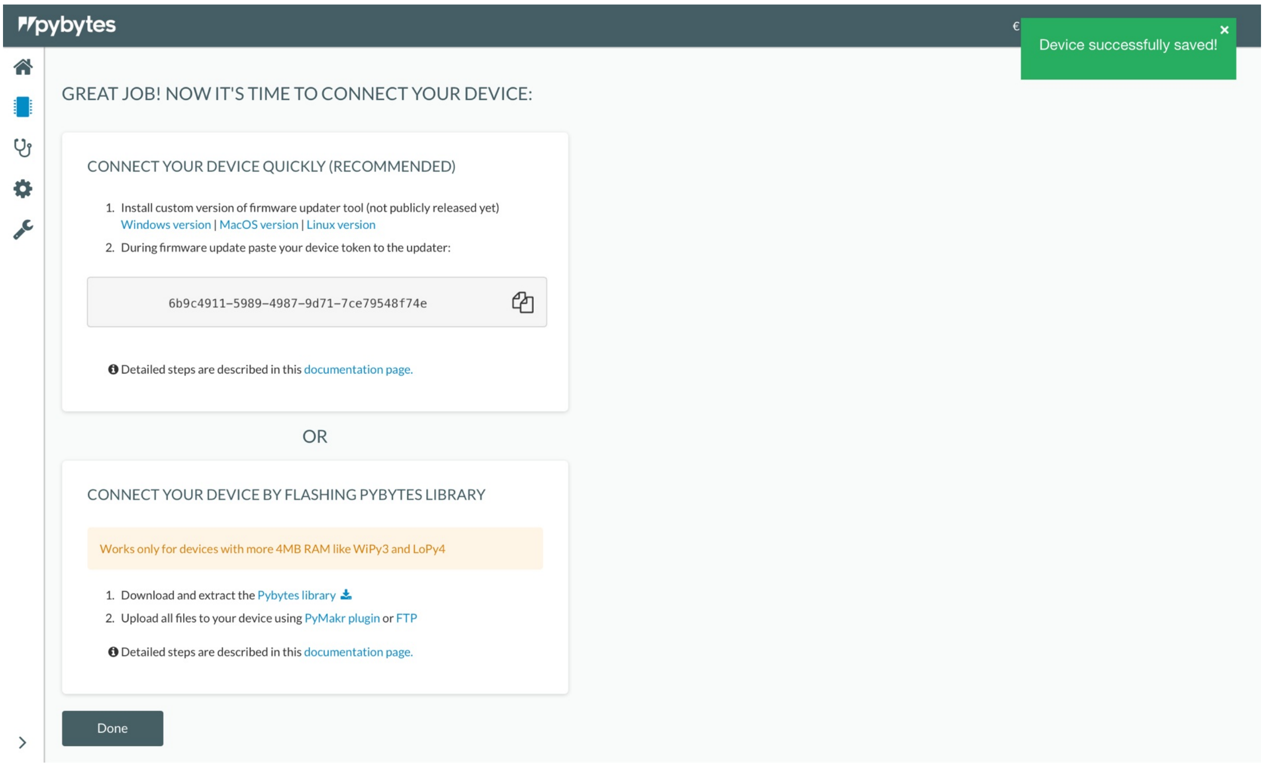


5. Enter a unique name and the network credentials (SSID and password) for your device;



Step 2: Connect your device to Pybytes

At the end of the "Add Device" wizard, Pybytes will give you two options for you to connect your device to Pybytes:



Select how you would like to connect your device to Pybytes:

1. CONNECT YOUR DEVICE *QUICKLY (RECOMMENDED)*
2. CONNECT YOUR DEVICE *BY FLASHING PYBYTES LIBRARY*

From firmware 1.16.x onwards all Pycom devices come with Pybytes library build-in `/frozen` folder. That means that you can choose between adding your device quickly with the firmware updater or you can flash Pybytes library manually.

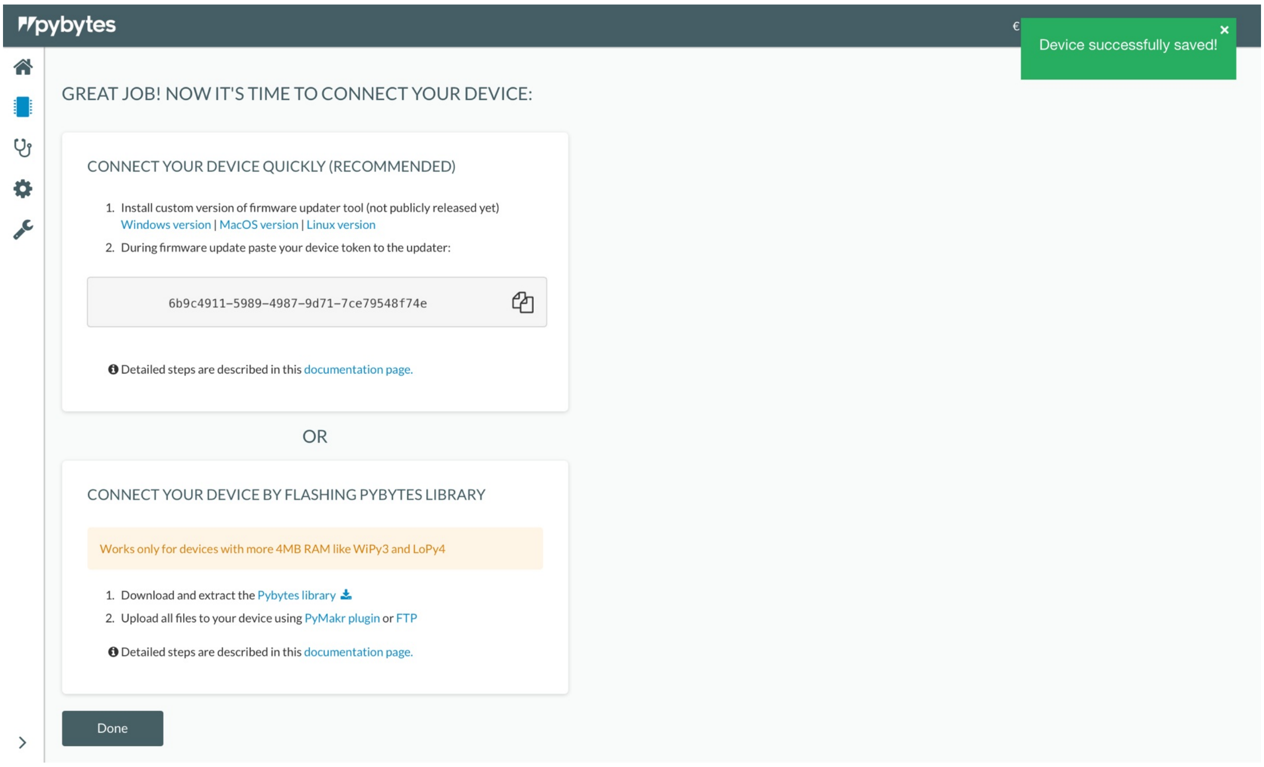
Connecting a device to Pybytes quickly by using the Firmware Updater

In this section, we explain to you how to connect your device to Pybytes quickly using the Firmware Updater.

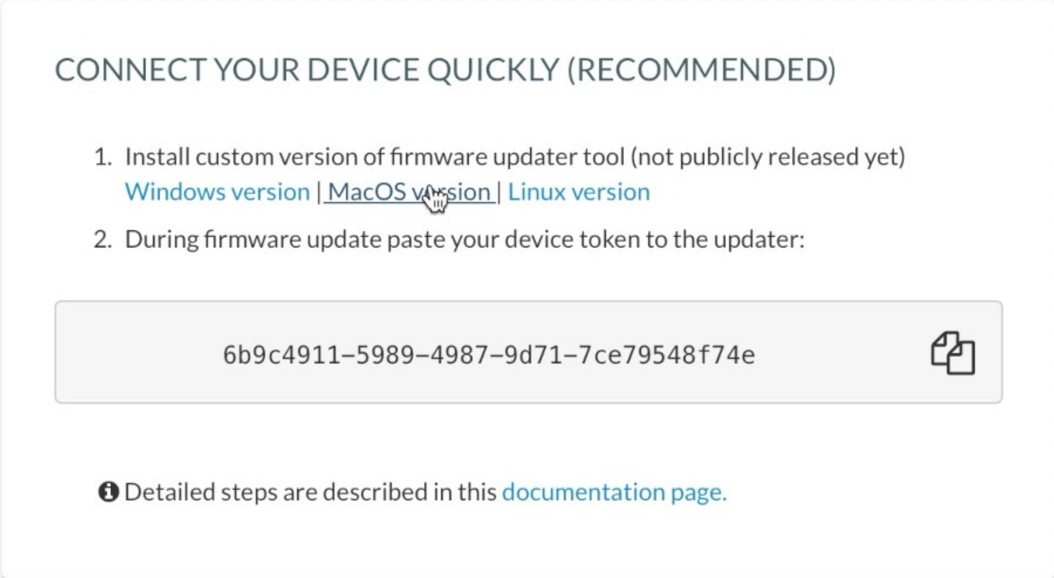
> In case you want to extend Pybytes library you can flash Pybytes library manually. Click here for more information.

Step 1: Download the firmware updater

At the last step of the "Add Device" process:



1. Download the firmware updater for your operating system;

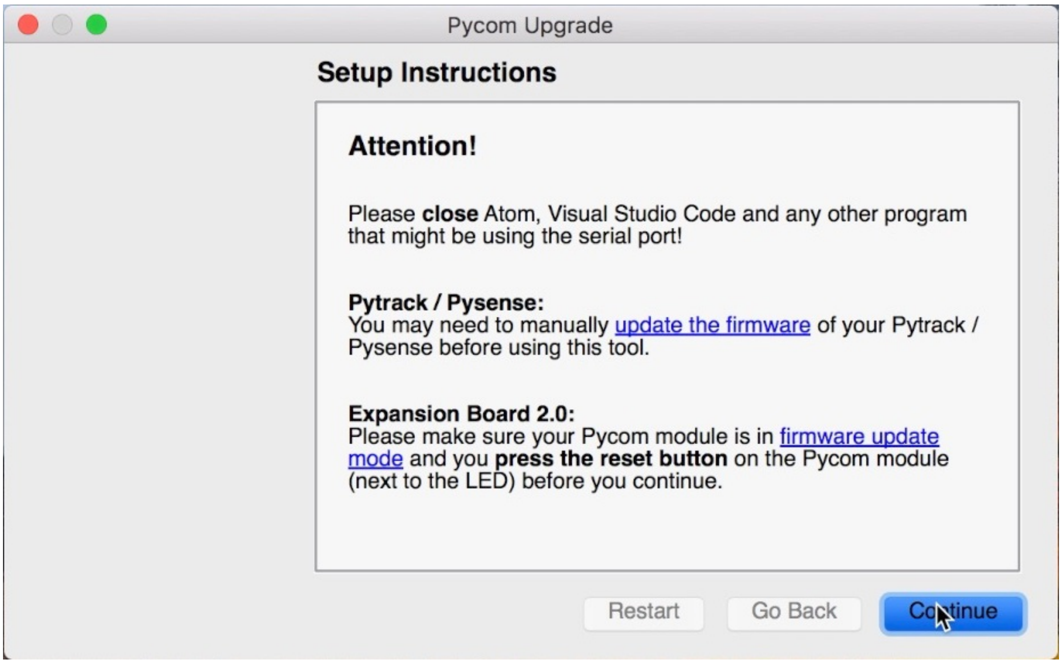


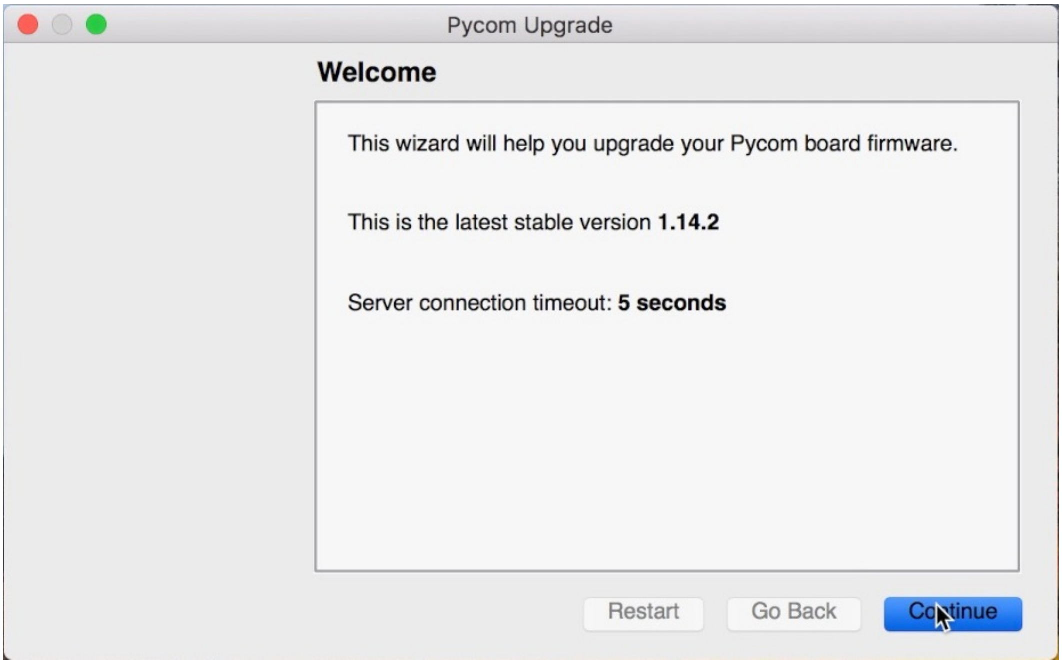
2. Copy the device token.

Step 2: Firmware updater

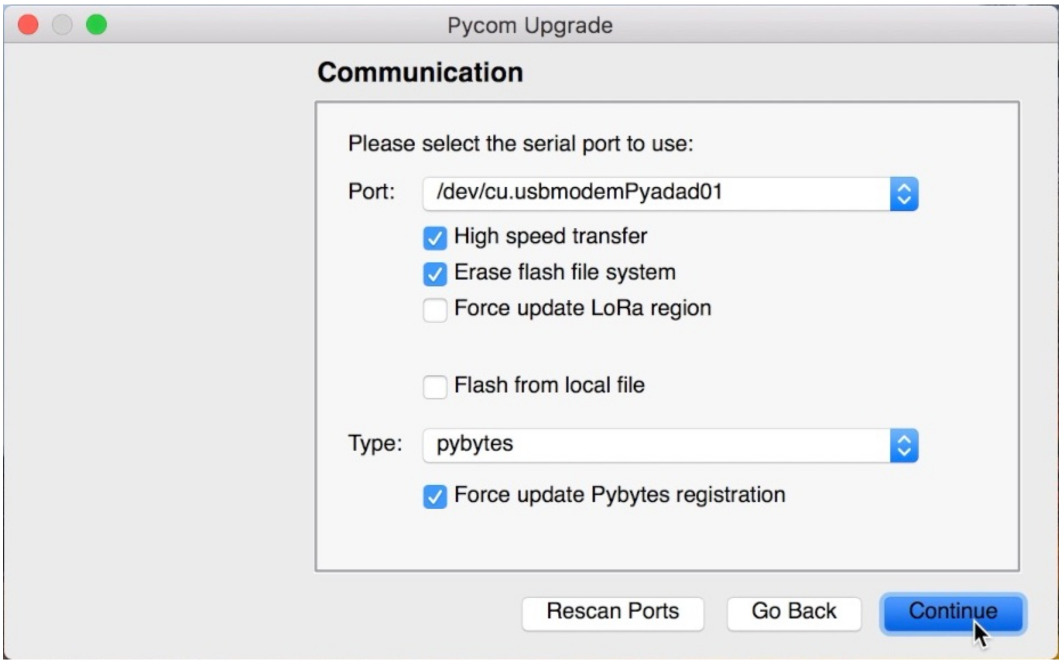
Install the Firmware updater on your computer.

1. Start the `Firmware updater` ;

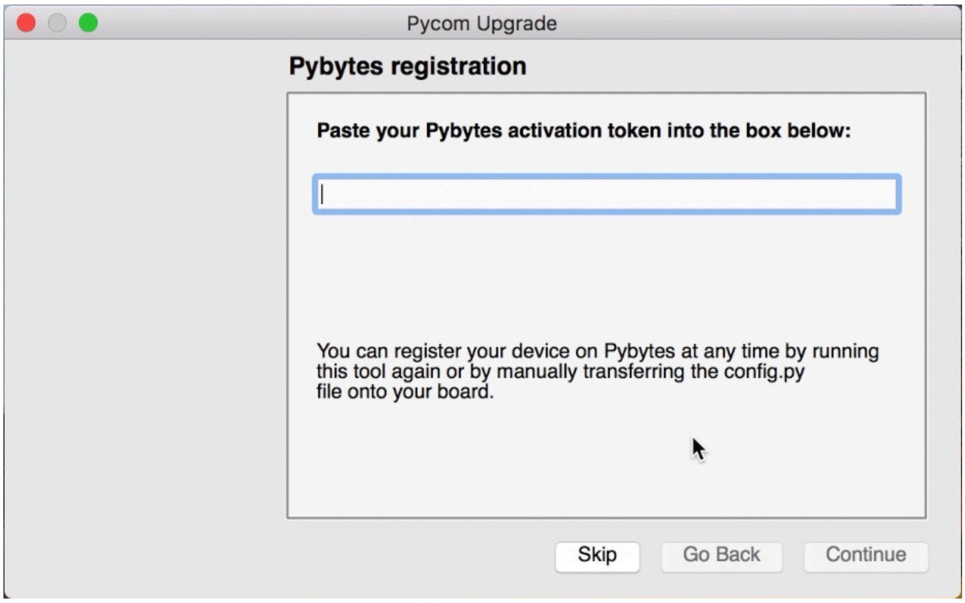




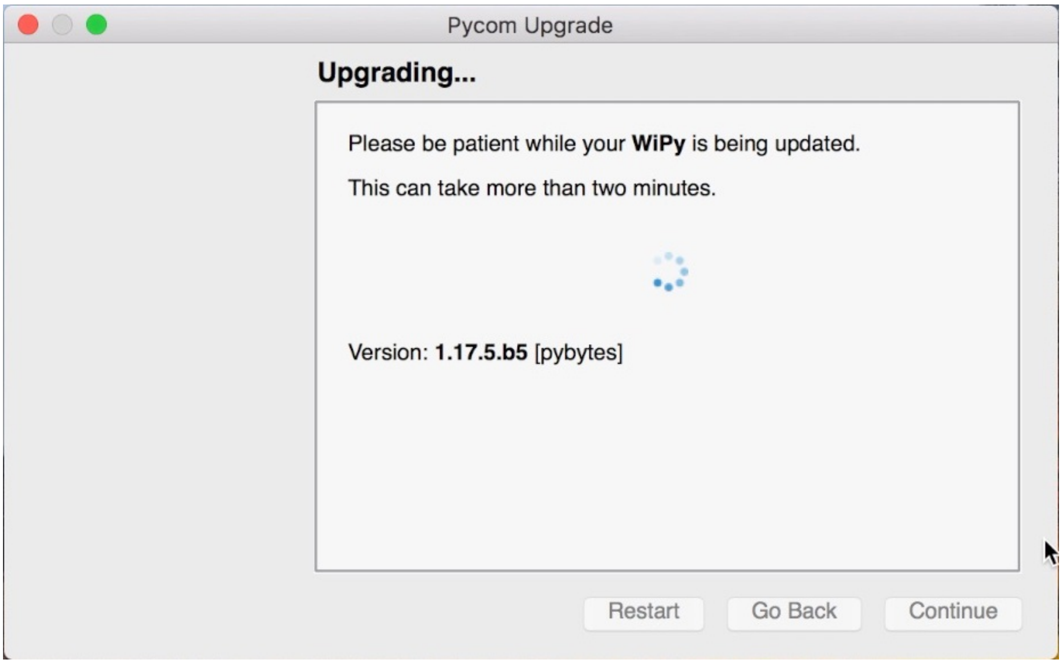
2. Select your device serial port (Make sure your device is connected to your computer);
3. Mark the options "Erase flash file system" and "Force update Pybytes registration";

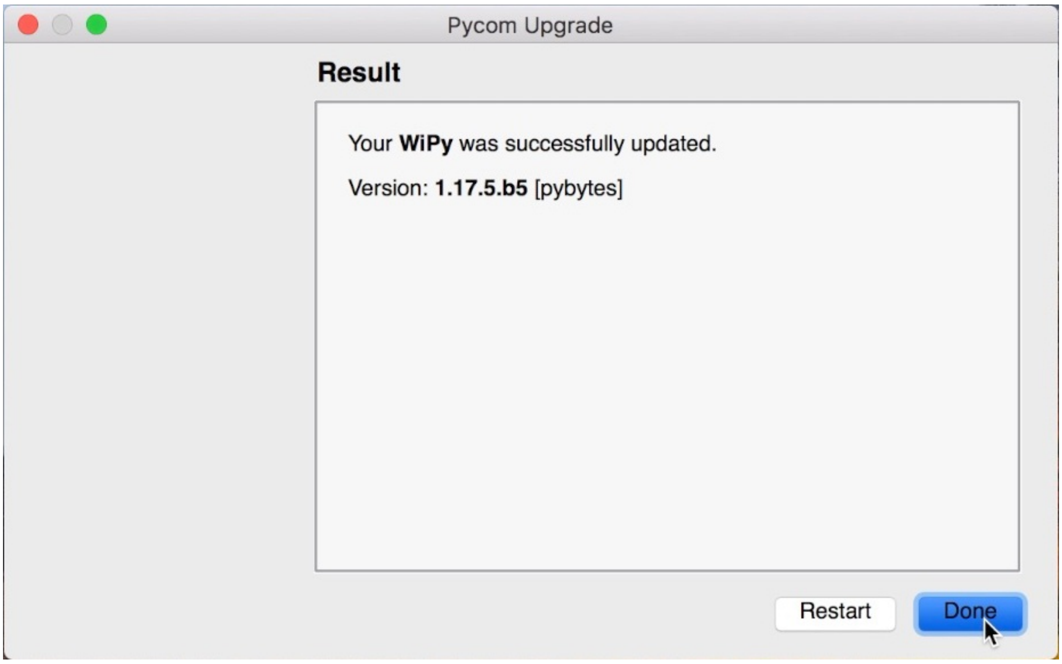


4. Paste your device token from Pybytes;



5. The firmware updater will update the device's firmware.





Next step: Set up your device's dashboard!

Now it's time to display data from your device into Pybytes dashboard. You can check more about it here!

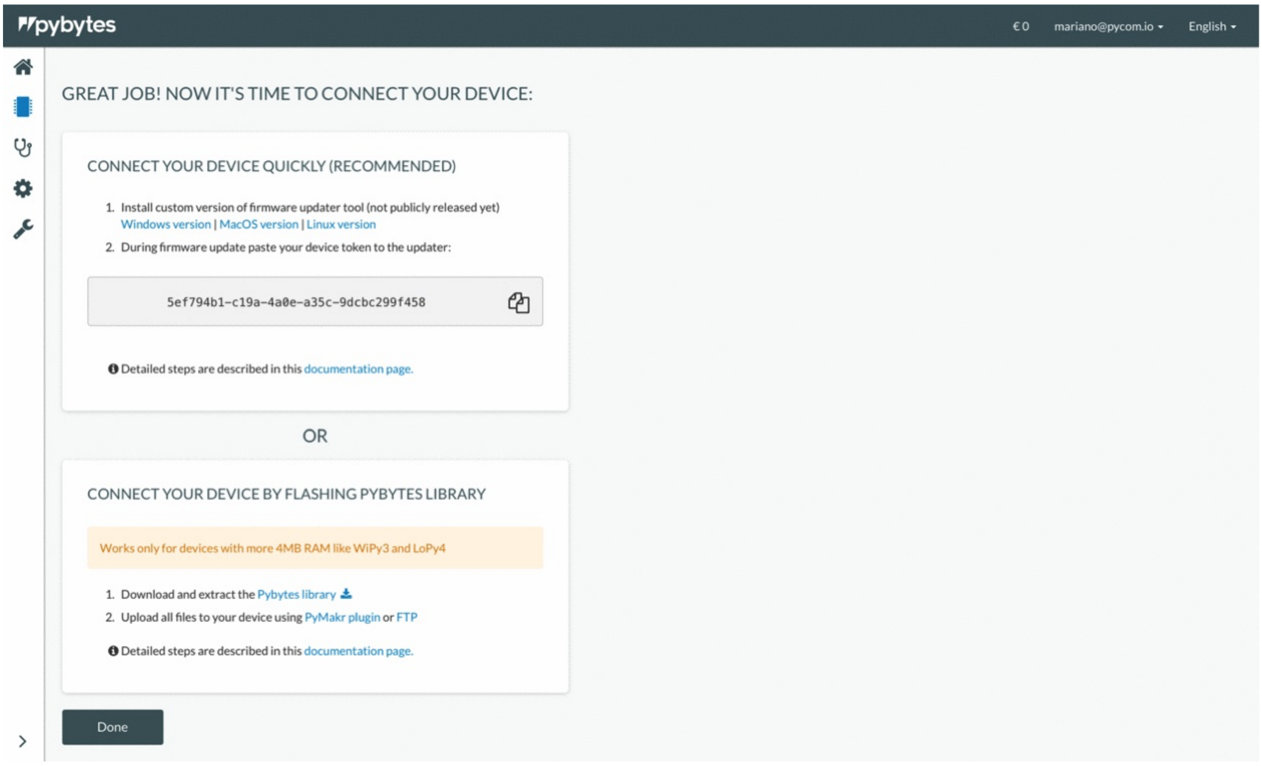
Connecting a device to Pybytes by flashing Pybytes library manually

In this section, we will explain to you how to connect your device to Pybytes by flashing Pybytes library manually.

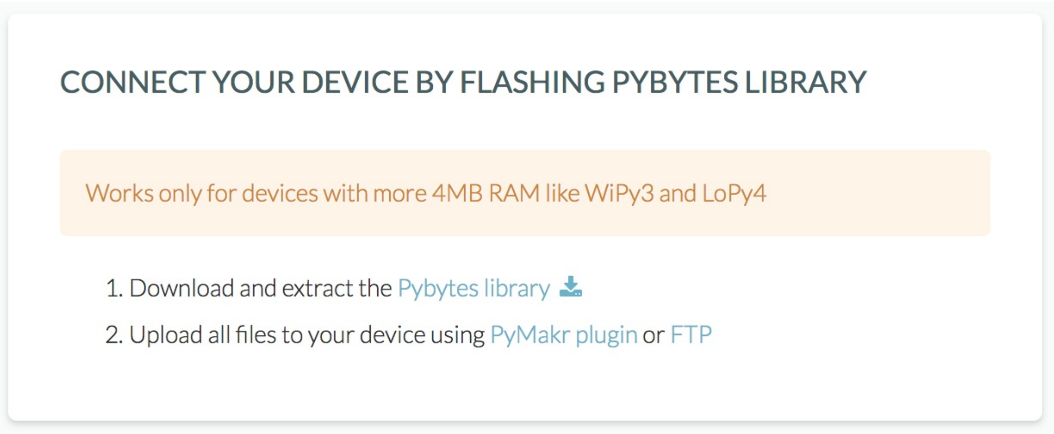
> From firmware 1.16.x onwards all Pycom devices come with Pybytes library build-in `/frozen` folder. That means that you can add your device quickly without the need of flashing Pybytes library manually. Click here for more information.

Step 1: Download your Pybytes Library

At the last step of the "Add Device" process:

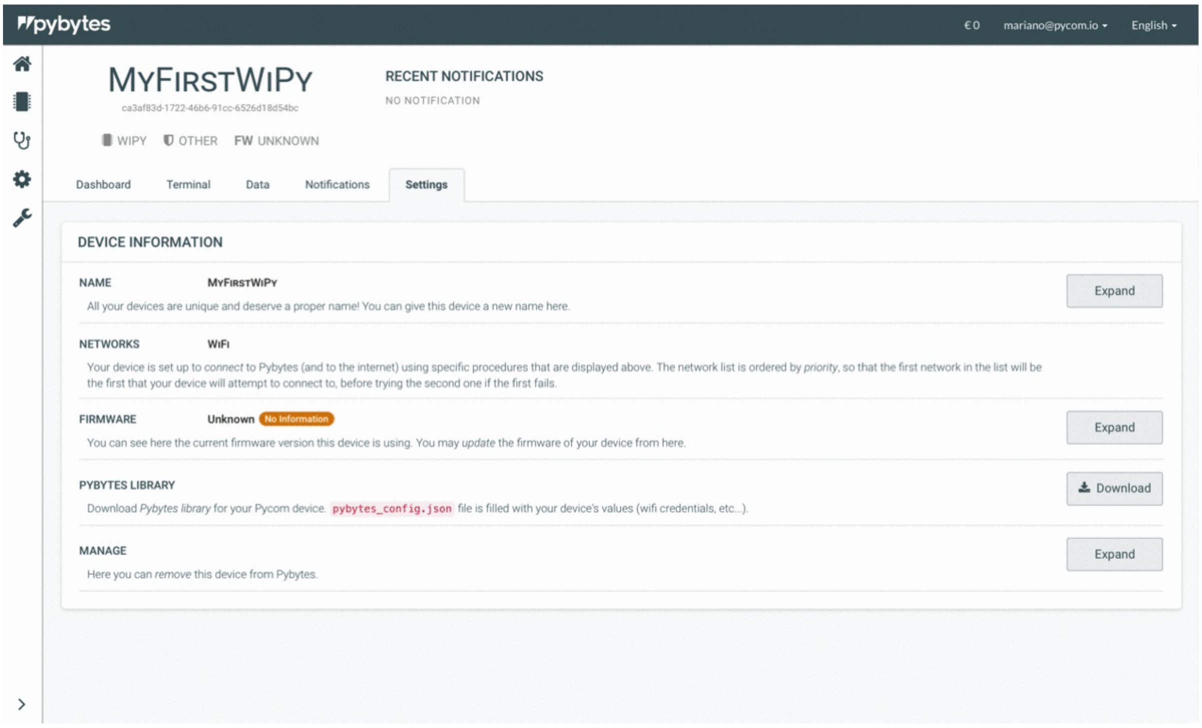


1. Click on download *Pybytes library*



You can also download *Pybytes library* at the device's settings page:

1. Navigate to your device in Pybytes;
2. On your device's page click on settings tab;
3. Click on the button *Download* at Pybytes library;



Step 2. Flash your device with Pymakr

In case you haven't installed Pymakr plugin, follow these instructions.

1. Connect your device to your computer with USB cable.
2. Extract download Pybytes library and open extracted folder with Atom.
3. Get your device serial port: in Pymakr plugin click on *More > get serial ports*
4. Paste your device's serial port to `pymakr.conf` file:

```
{
    "address": "PASTE_YOUR_SERIAL_PORT_HERE",
    "username": "micro",
    "password": "python",
    "sync_folder": "flash"
}
```

5. Checkout your `flash/pybytes_config.json` file. It will be pre-filled with your information from Pybytes Like deviceToken or WiFi credentials. You can change e.g. your WiFy password here.
6. Put your device in safe boot mode.
7. Upload code to your device by clicking on *Upload* button in Pymakr. After all Pybytes library files are uploaded to device, device will restart and will connect to Pybytes.

Pybytes library is written to `/flash` folder and will take precedence over build in firmware libraries in `/frozen` folder.

Next step: Set up your device's dashboard!

Now it's time to display data from your device into Pybytes dashboard. You can check more about it here!

Visualise data from your device.

In this section, we will explain to you how to create widgets for data visualisation and set up your device's dashboard on Pybytes.

> We assume that you already have your device connected to Pybytes. In case you haven't, check how to add your device here. After your done with that, you can proceed to the next example.

Step 1: Set up your application (main.py)

The first step is to have an application running on your device. The application in this example sends data from a vector every 10 seconds to Pybytes.

1. Open the `main.py` file on Pymakr;
2. Insert the following code on your `main.py` ;

```
# # Import what is necessary to create a thread
import _thread
from time import sleep

# # Increment index used to scan each point from vector sensors_data
def inc(index, vector):
    if index < len(vector)-1:
        return index+1
    else:
        return 0

# # Define your thread's behaviour, here it's a loop sending sensors data every 10 sec
onds
def send_env_data():
    idx = 0
    sensors_data = [0, -0.2, -0.5, -0.7, -0.8, -0.9, -0.9, -0.9, -0.8, -0.6, -0.4, -0.2
, 0, 0.3, 0.5, 0.7, 0.8, 0.9, 0.9, 0.9, 0.8, 0.6, 0.4, 0.1]

    while (pybytes):
        pybytes.send_virtual_pin_value(False, 1, sensors_data[idx])
        idx = inc(idx, sensors_data)
        sleep(10)

# # Start your thread
_thread.start_new_thread(send_env_data, ())
```

1. Upload the code into your device. Now your device is sending data to Pybytes.

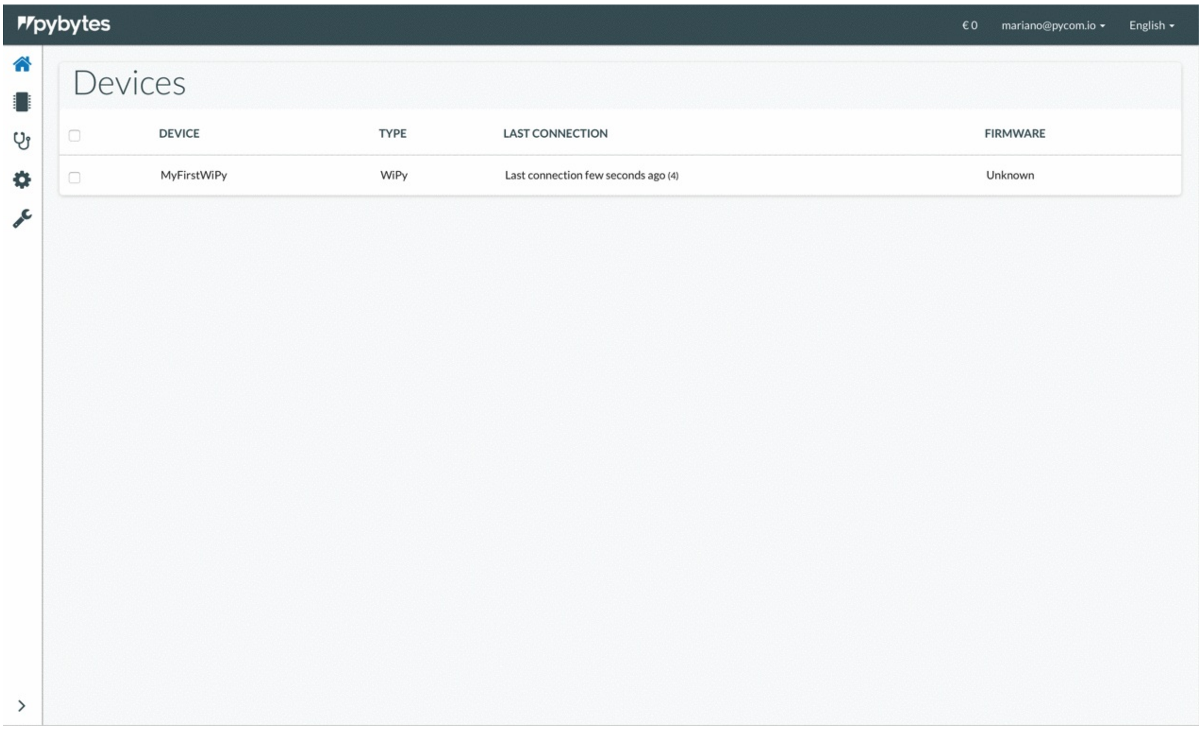
   In this code, we're calling the function `pybytes.send_virtual_pin_value(persistent, pin, value))` to communicate with Pybytes. This function is part of the Pybytes library, and it has three arguments: `persistent`, `pin` and `value`.

   - `persistent` denotes information that is infrequently accessed and not likely to be modified;
   - `pin` represents which virtual pin is receiving data;
   - `value` is the value being attributed to that particular pin.

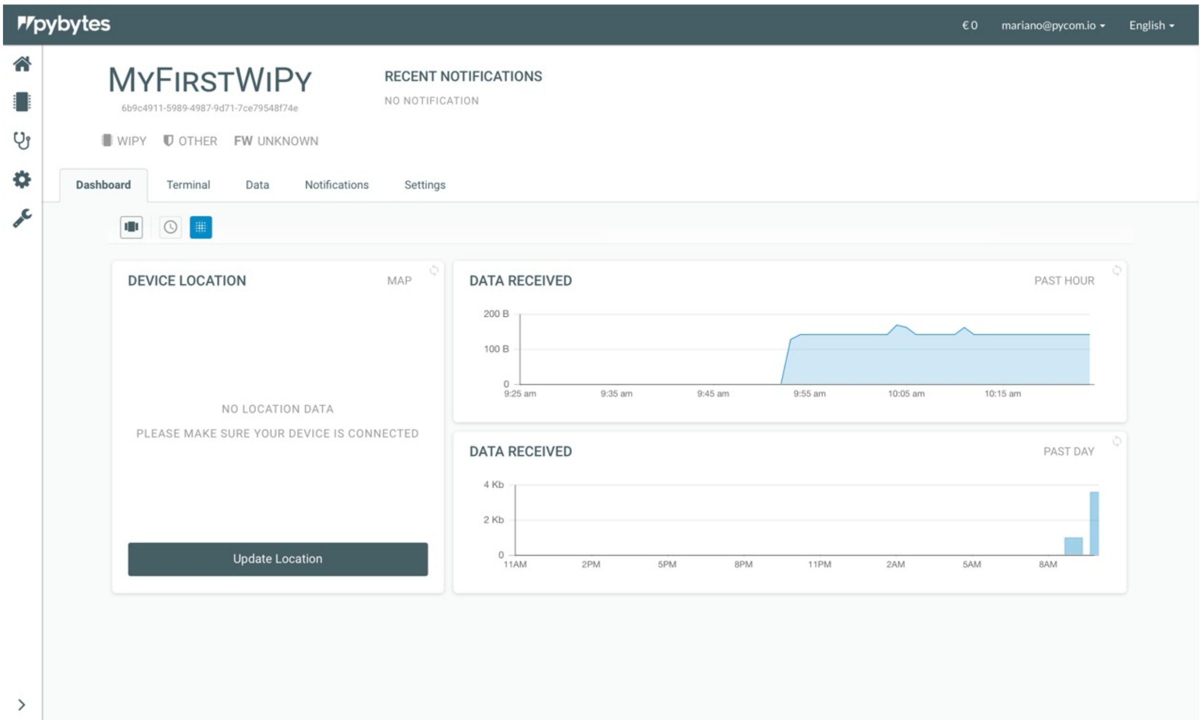
Step 2: Add a signal from your device

Go to Pybytes.

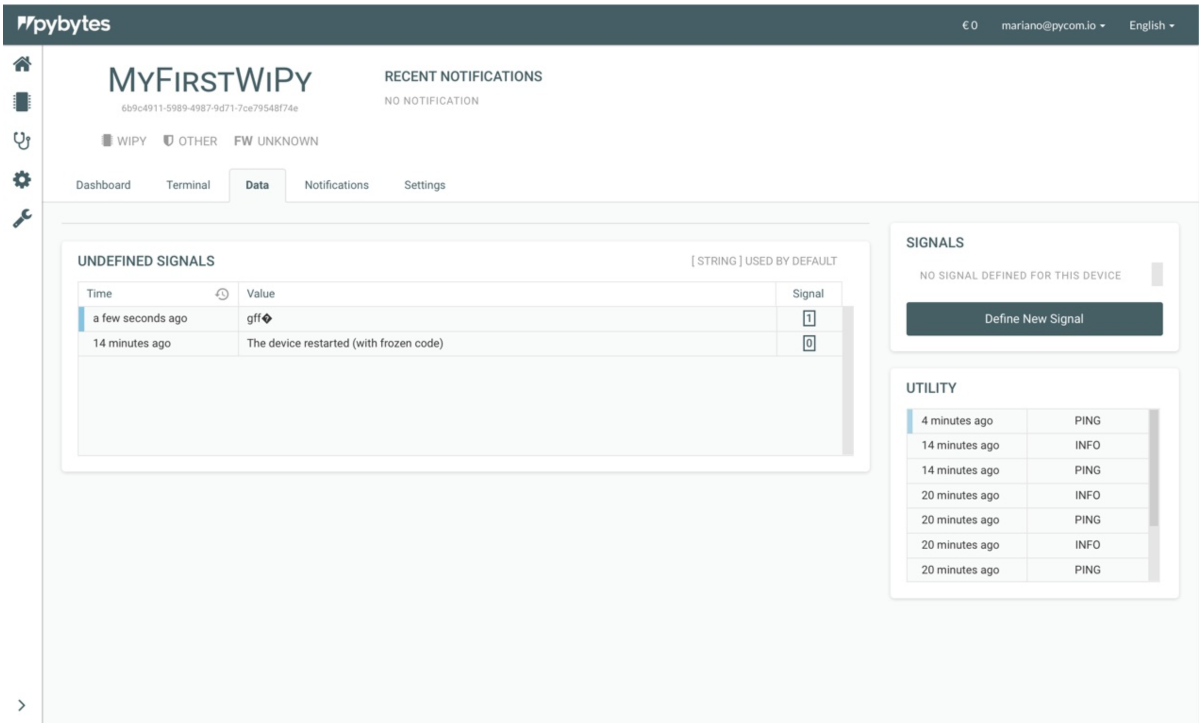
1. On `Devices` page select a device;



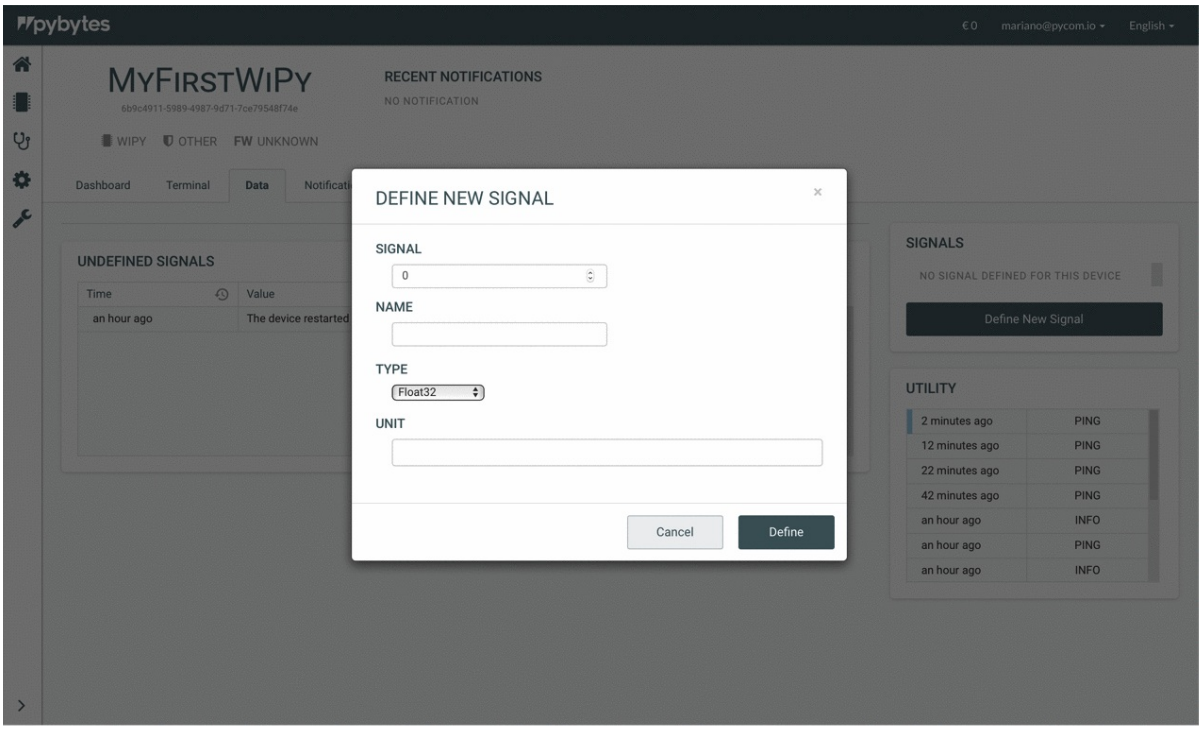
2. On your device's page click on `Data` tab.



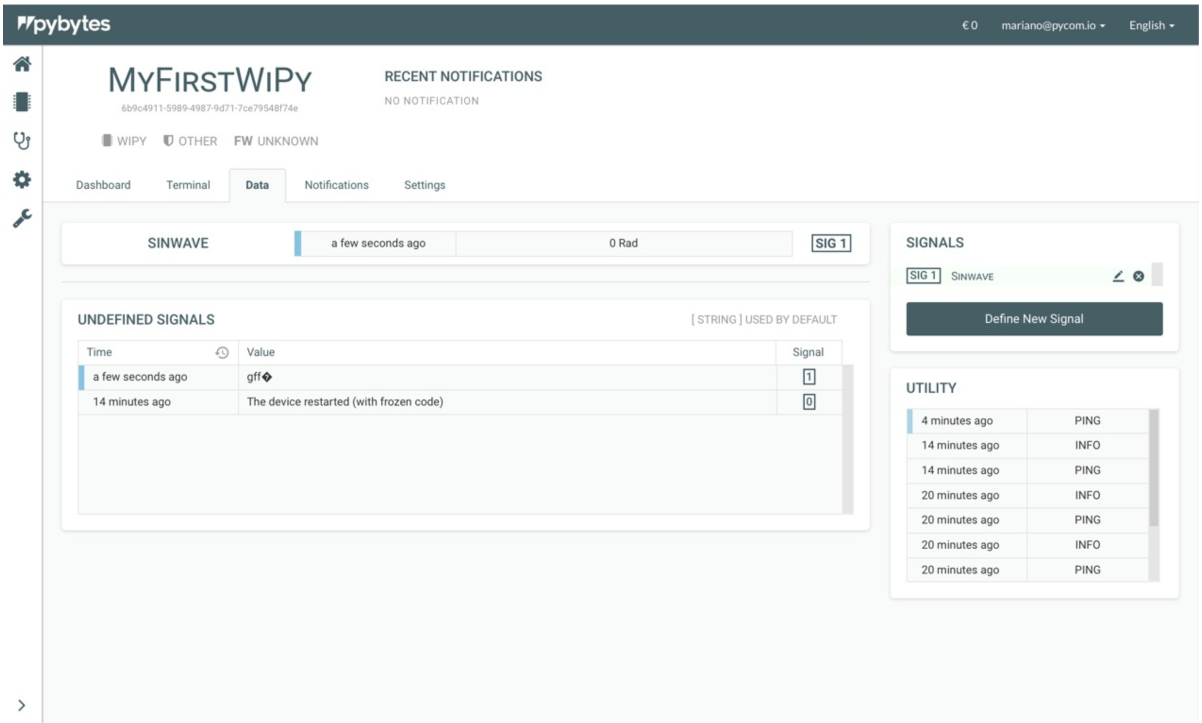
3. Click on the `Define New Signal` button.



4. Define the new signal by entering a number, a name, a data type and a unit. Finally, click on the button `Define` .



5. Your signal was added!



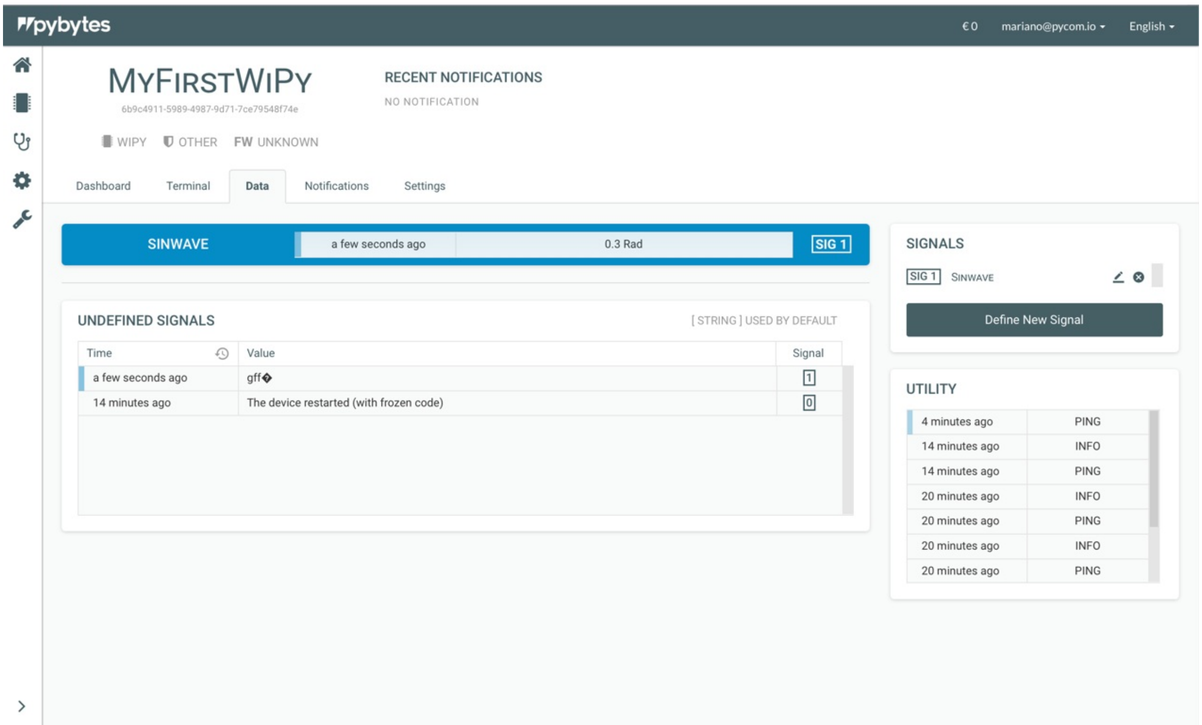
The name and unit are labels used to identify your signal inside Pybytes (In this example we defined `Sinwave` as the name of the signal and `Rad` as the unit).

The signal number has to match the pin number that you defined on `pybytes.send_virtual_pin_value` function call, inside your `main.py` code (In this example we defined `pin = 1`);

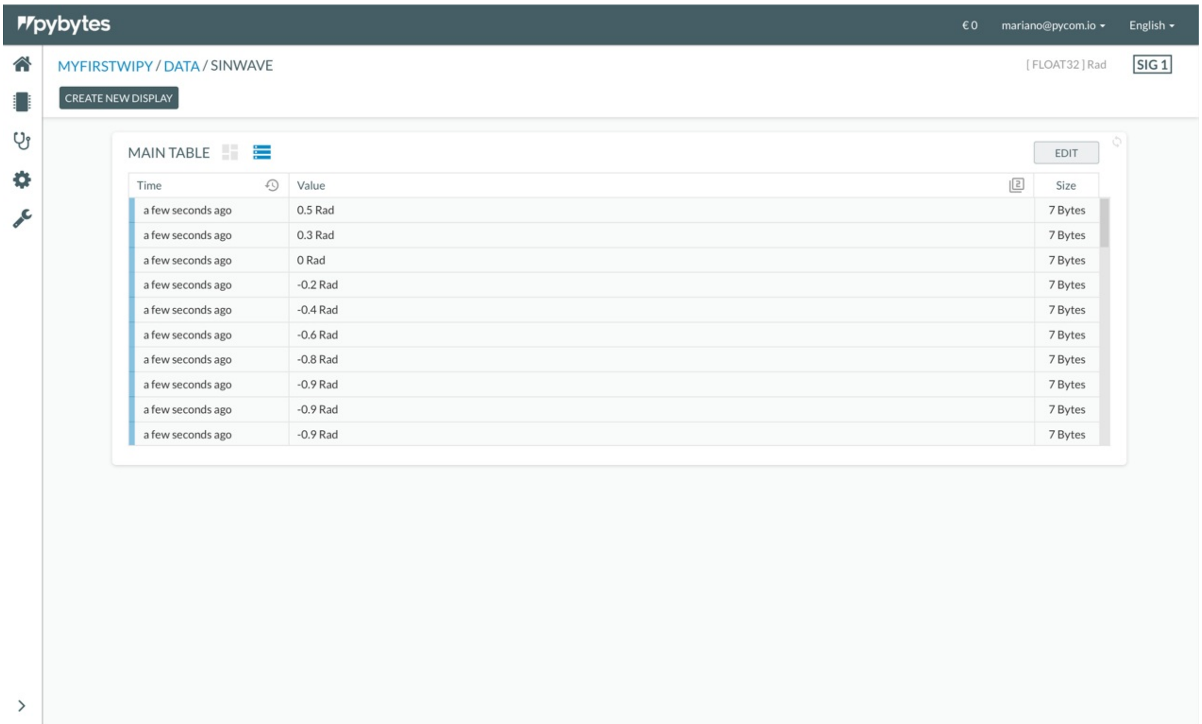
The datatype also has to match the variable used as argument on `pybytes.send_virtual_pin_value` function call, inside your `main.py` code (In this example our variable is a floating number; therefore we defined as a `Float32`).

Step 3: Add a widget for the signal

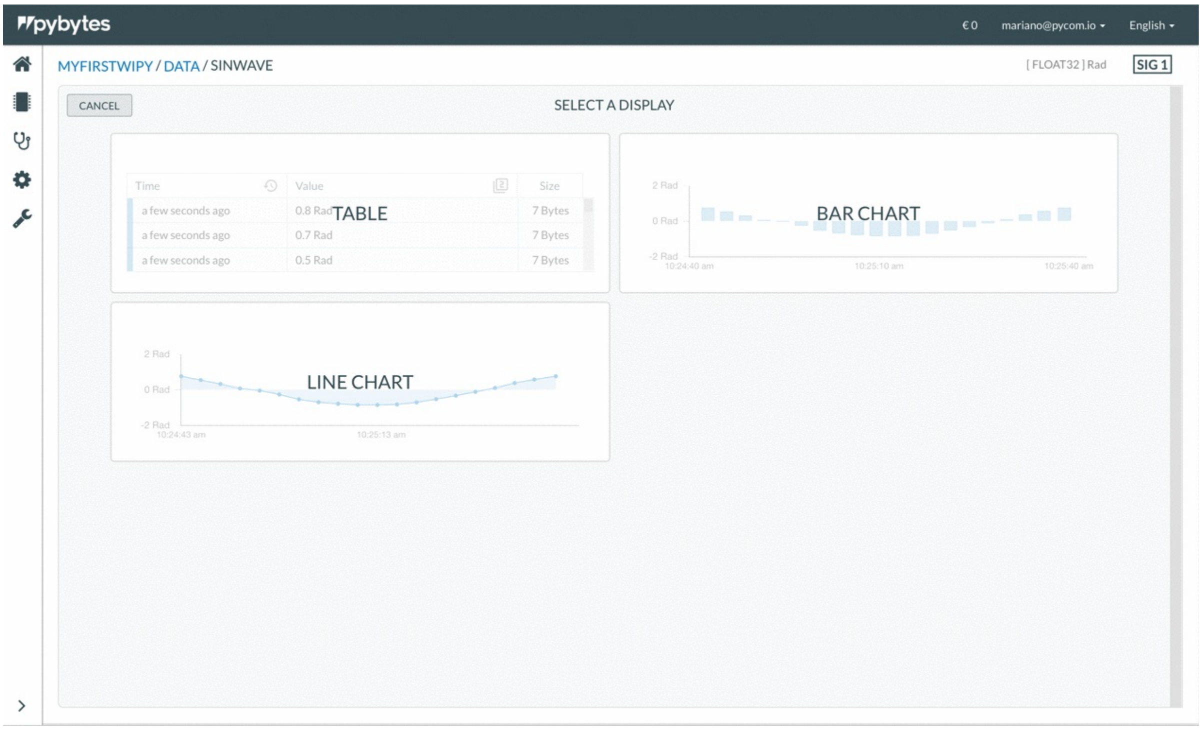
1. Click on the signal card.



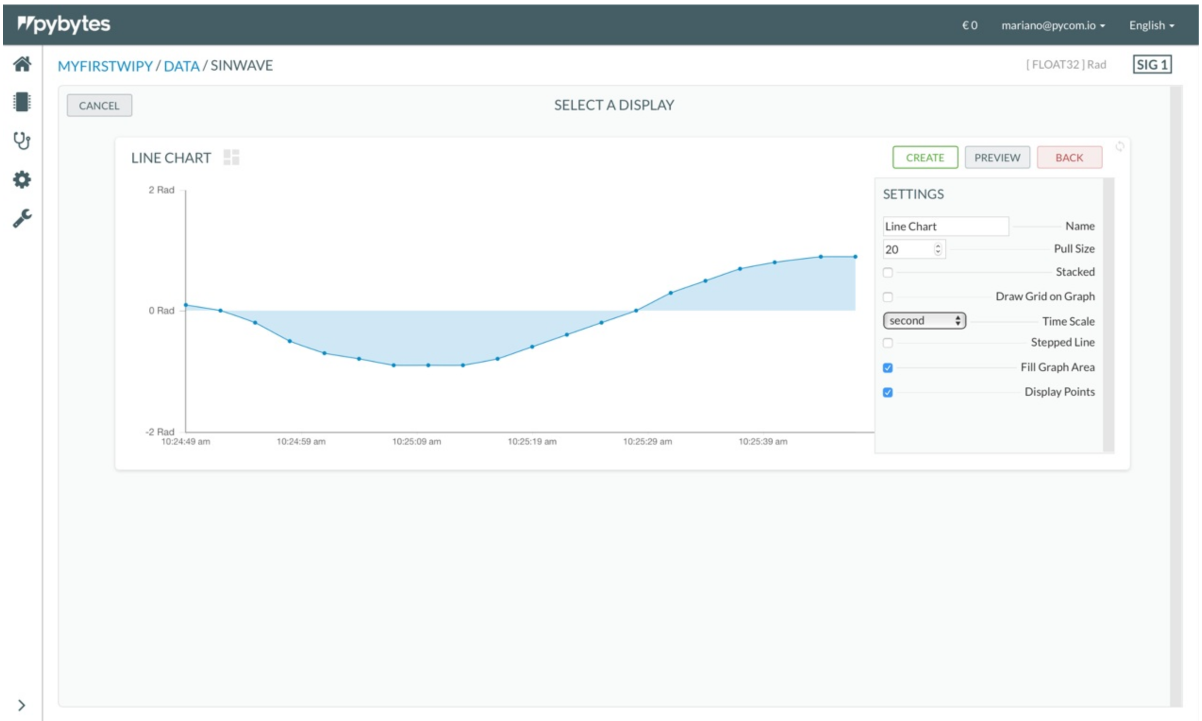
2. Click on the button `Create a new display`.



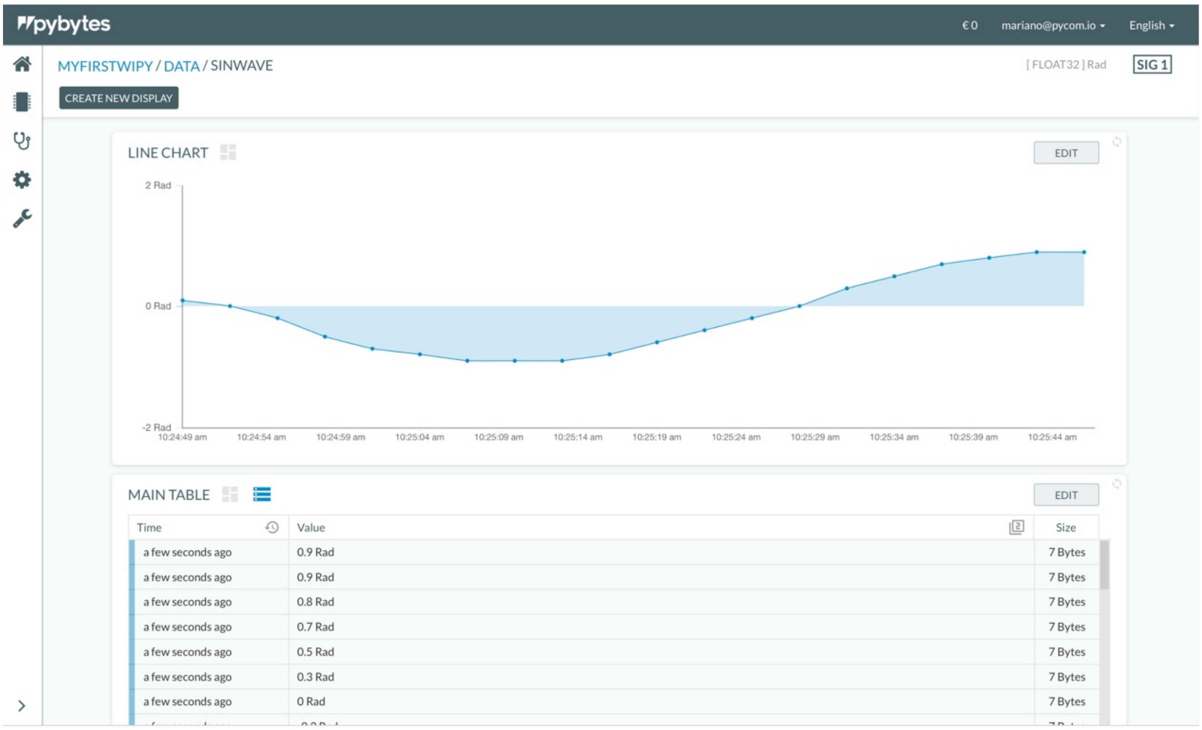
3. Select the type of visualisation (e.g. Bar chart or Line chart).



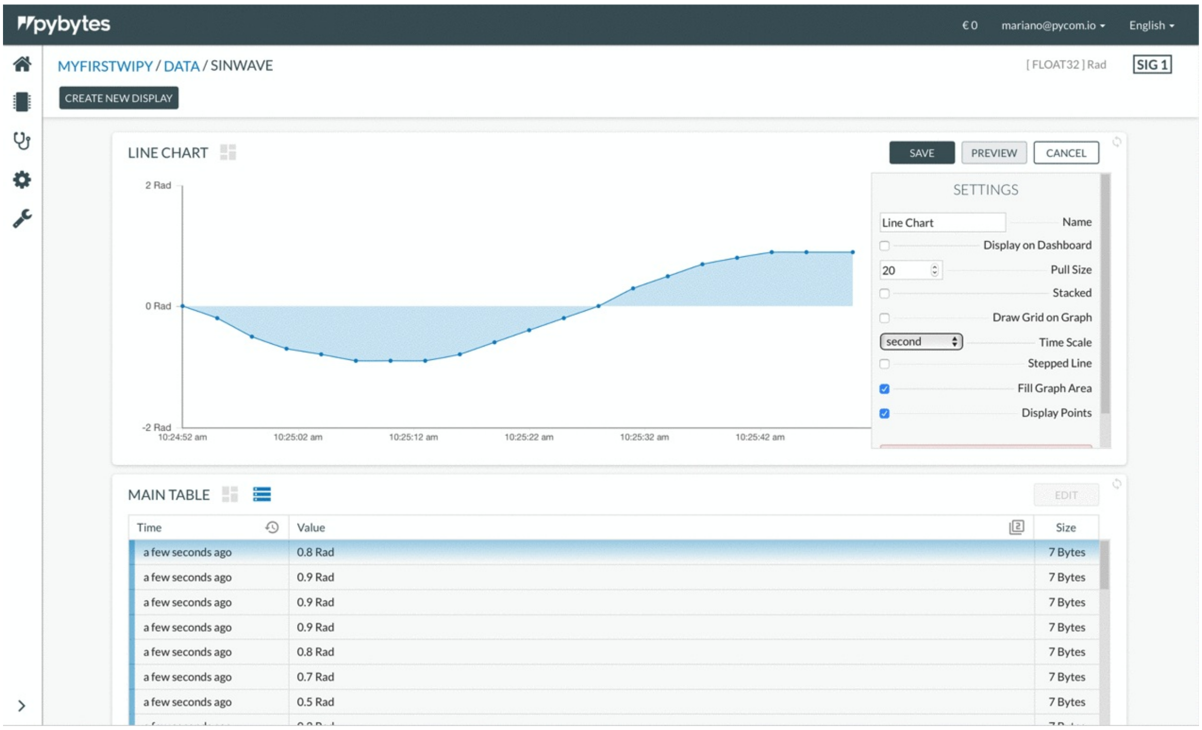
4. You can adjust the parameters of your widget at `Settings`. After, click on the button `Create`.



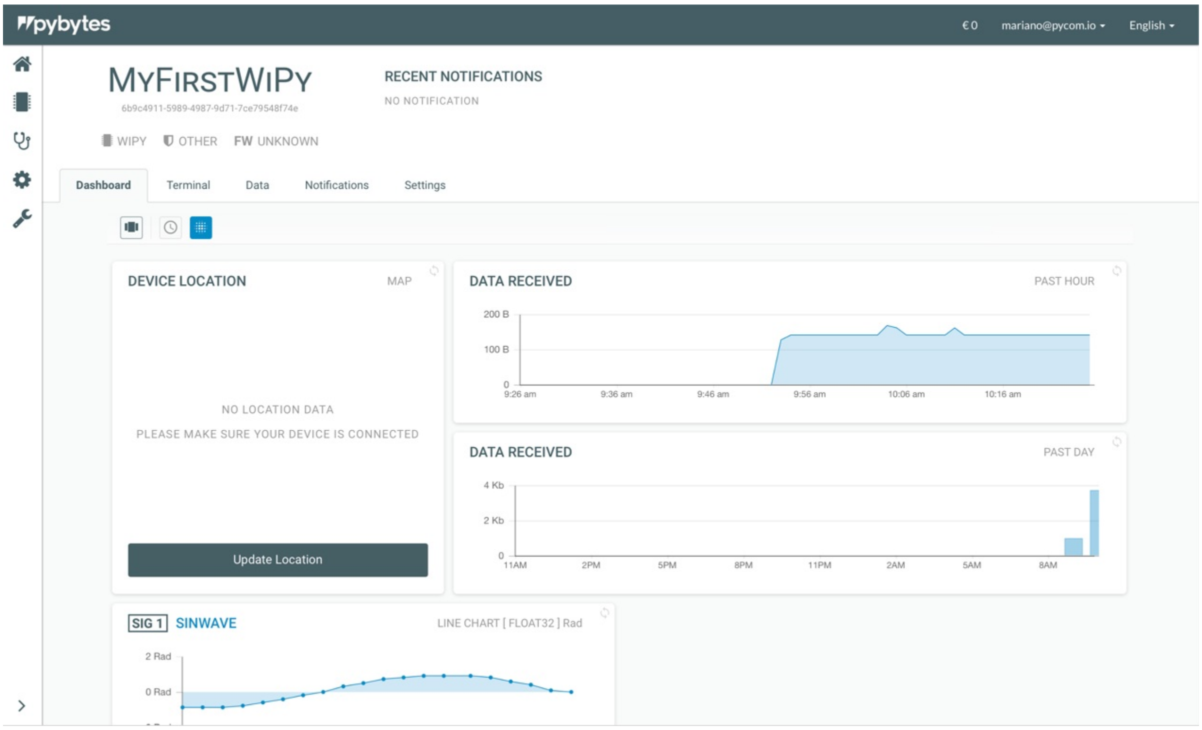
5. Your widget was created. Now, add your widget to your device's dashboard. Click on the button `Edit` on your widget.



6. Mark the checkbox `Display on Dashboard` at `Settings`. Finally, click on the button `Save`.

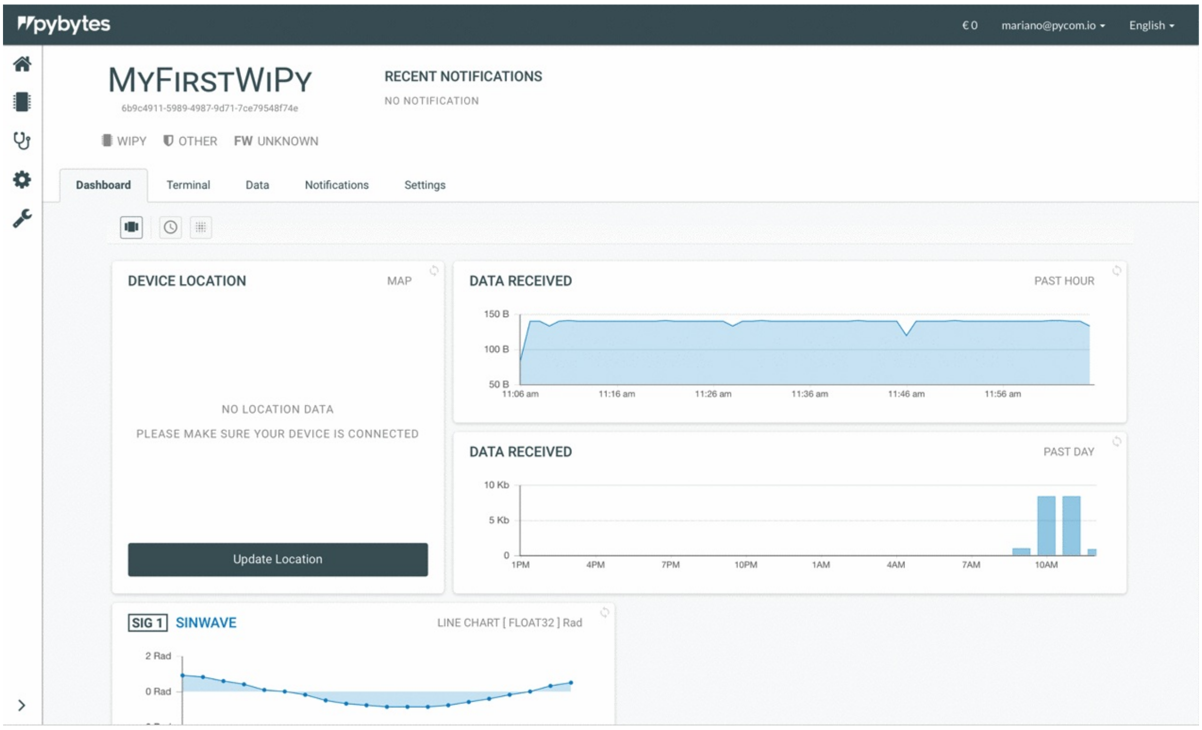


7. Click on the tab `Dashboard`. Your widget was successfully added there!

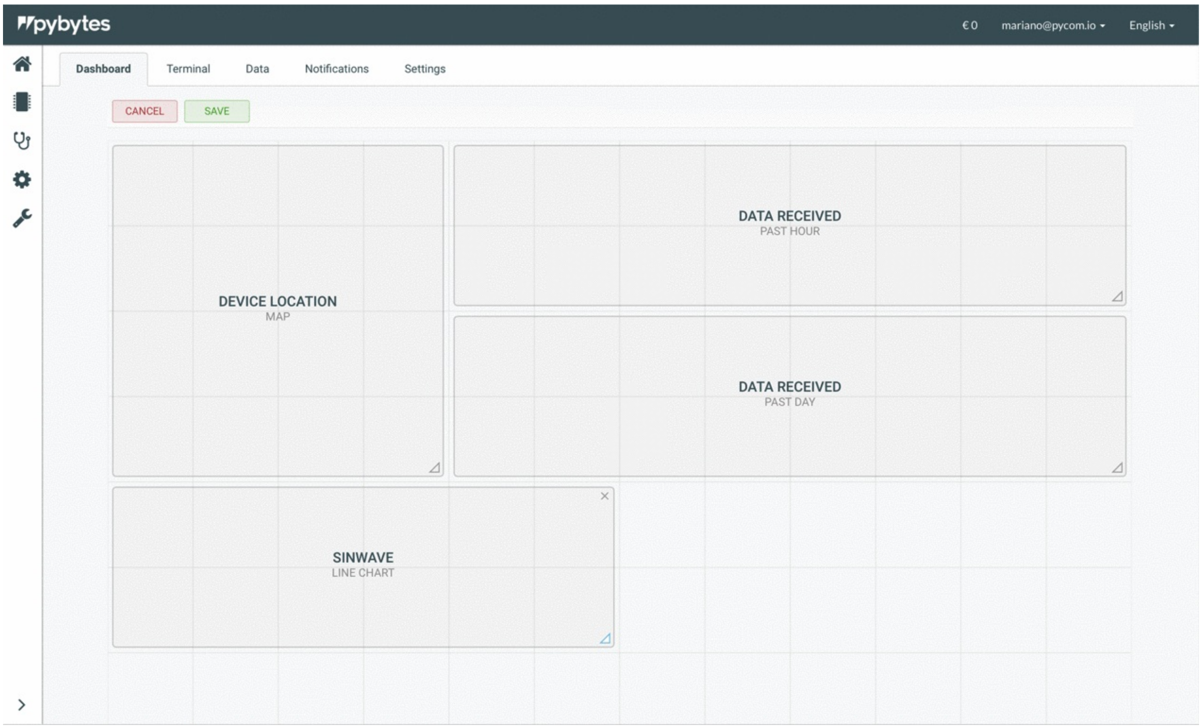


Step 4: Organise your dashboard

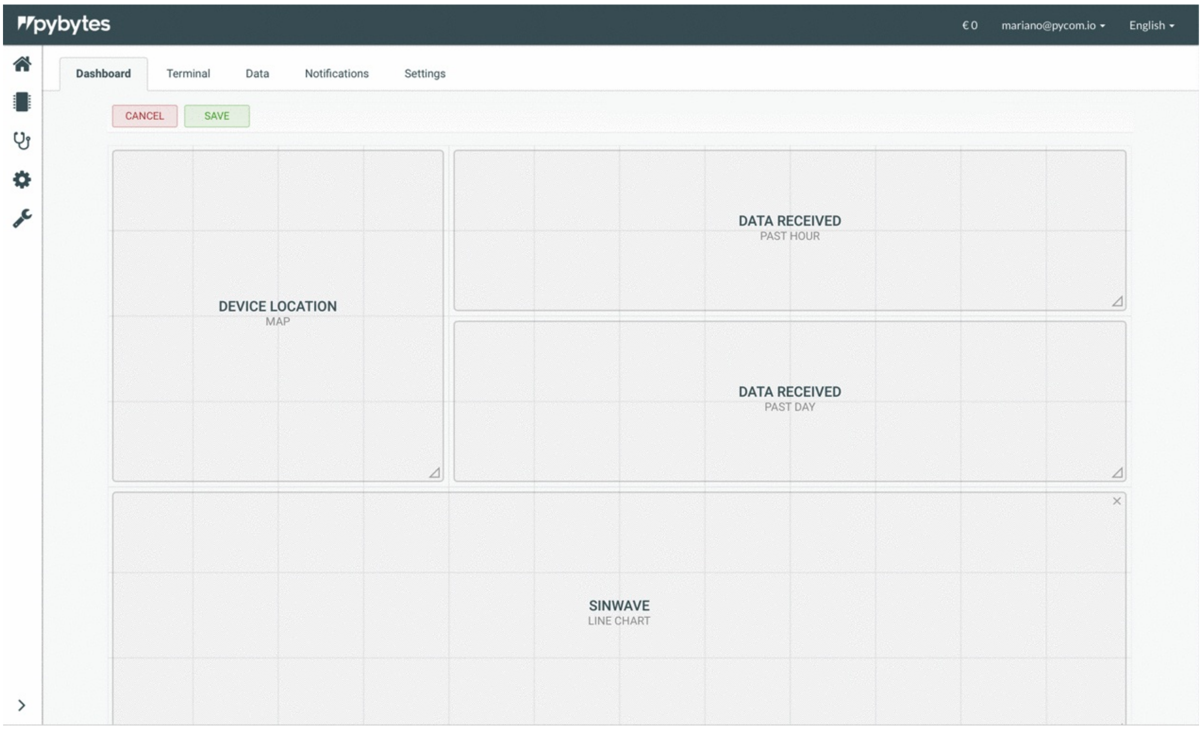
1. Click on the button `Organise`. Now the dashboard's grid will enter the edit mode and allow you to resize and reposition its widgets.



2. Resize a widget by clicking on the triangle icon at the bottom right corner of the widget and drag the cursor over the grid. After, click on the button `Save` to save this action.



3. Change the widget's position by drag-and-dropping it over the grid. After, click on the button `Save` to save this action.



Done!

Now you've learned how to set up your device's dashboard to display data. Also, you can add more widgets to other pins of your device.

Documentation Notes

The Pycom documentation aims to be straightforward and to adhere to typical Python documentation to allow for ease of understanding. However, there may be some unusual features for those not used to Python documentation or that are new to the MicroPython Language. This section of the documentation aims to provide clarity for any of the design specifics that might be confusing for those new to Python and this style of documentation.

Documentation Syntax

The Pycom documentation follows standard Python Library format using the popular Sphinx Docs tool. There are some notable points regarding the syntax of classes, methods and constants. Please see the notes below and familiarise yourself with the specific details before reviewing the documentation.

Keyword Arguments

`Keyword Arguments` refer to the arguments that are passed into a constructor (upon referencing a class object). When passing values into a MicroPython constructor it is not always required to specify the name of the argument and instead rely on the order of the arguments passed as to describe what they refer to. In the example below, it can be seen that the argument `mode` is passed into the `i2c.init()` method without specifying a name.

The values of the arguments (as seen in the examples/docs) refer to the default values that are passed into the constructor if nothing is provided.

i2c.init(mode, * , baudrate=100000, pins=(SDA, SCL))
An example of how this method might be called:

```
i2c.init(I2C.MASTER, pins=('P12', 'P11'))
```

It can be seen that a value for `baudrate` was not passed into the method and thus MicroPython will assume a default value of `100000`. Also the first argument `mode` was not specified by name, as the constructor does not require it, denoted by the lack of an `=` symbol in the constructor documentation.

Passing Arguments into a Method

It is important to note that there are certain class methods that can only accept a `keyword` for certain arguments as well as some that only accept a `value`. This is intentional by design but is not always apparent to the user calling specific methods. The differences between the two are outlined below, with examples referencing where differences might apply and what to be aware of.

Keyword

An astrik `*` in a method description (in the docs), denotes that the following arguments require a keyword, i.e. `pin='P16'` in the example below.

adc.channel(* , pin, attn=ADC.ATTN_0DB)

```
from machine import ADC

adc = ADC()                      # create an ADC object
apin = adc.channel(pin='P16')    # create an analog pin on P16
```

`pin` is a required argument and the method `channel` will not execute unless it is passed as with a keyword.

Another example shows how the `PWM` class, `pwm.channel()` requires a keyword argument for `pin` but does not for `id`.

```
from machine import PWM

pwm = PWM(0, frequency=5000)
pwm_c = pwm.channel(0, pin='P12') # no keyword argument requires for id (0) but is req
uired for pin (pin='P12')
```

Value

The documentation may refer to a method that takes an argument listed by name but does allow for a keyword to be passed. For example, the `pycom` class contains a method `rgbled`. This lists that the method accepts a value for `color`, however this may not be specified by `keyword`, only `value`. This is intentional as the `value` being passed is the only argument valid for this method

pycom.rgbled(color)
If the argument is passed into the method with a keyword, it will return an error stating TypeError: function does not take keyword arguments.

```
import pycom

pycom.rgbled(color=0xFF0000) # Incorrect
pycom.rgbled(0xFF0000) # Correct
```

Another example of a method that only accepts value input. In this case, the `RTC.init()` method require a value (`tuple`) input for the `datetime`. It will not accept a keyword.

rtc.init(datetime)

```python
from machine import RTC

rtc = RTC()
rtc.init(datetime=(2014, 5, 1, 4, 13, 0, 0, 0)) # Incorrect
rtc.init((2014, 5, 1, 4, 13, 0, 0, 0)) # Correct
```

Constants

The `constants` section of a library within the docs refers to specific values from that library's class. These might be used when constructing an object from that class or when utilising a method from within that class. These are generally listed by the library name followed by the specific value. See the example below:

I2C.MASTER()

> Be aware that you can only reference these constants upon importing and constructing a object from a library.

REPL vs Scripts

Users of this documentation should be aware that examples given in the docs are under the expectation that they are being executed using the MicroPython REPL. This means that when certain functions are called, their output may not necessarily be printed to the console if they are run from a script. When using the REPL many classes/functions automatically produce a printed output displaying the return value of the function to the console. The code snippet below demonstrates some examples of classes/functions that might display this behaviour.

Basic Arithmetic

```
1 + 1 # REPL will print out '2' to console
1 + 1 # Script will not return anything the console
print(1 + 1) # Both the REPL and a script will return '2' to the console
```

Calling Methods

```
import ubinascii

ubinascii.hexlify(b'12345') # REPL will print out "b'3132333435'" to the console
ubinascii.hexlify(b'12345') # Script will not return any the console
```

In order to use these functions that do not print out any values, you will need to either wrap them in a `print()` statement or assign them to variables and call them later when you wish to use them.

For example:

```
# immediately print to console when using a script
print(1 + 1)
# or save variable to for later
value = 1 + 1
# do something here...
print(value)
```

Firmware Downgrade

The firmware upgrade tool usually updates your device to the latest available firmware version. If you require to downgrade your device to a previous firmware there are two methods to achieve this.

> If you are using an Expansion Board 1.0 or 2.0, you will need to have a jumper connected between `G23` and `GND` to use either procedure below. You will also need to press the reset button before beginning.

You can obtain previous firmware versions here:

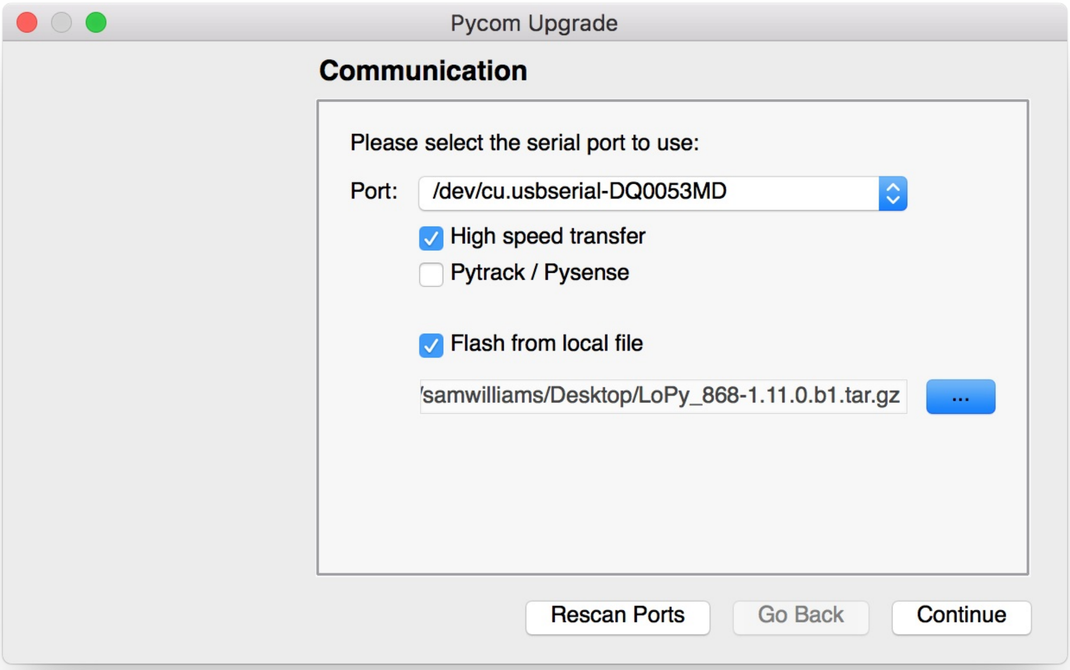
- WiPy
- LoPy
- SiPy
- GPy
- FiPy
- LoPy4

Note: Prior to version `1.16.0.b1` the firmware for modules with LoRa functionality was frequency specific. From `1.16.0.b1` and onward, the firmware is region agnostic and this can either be set programatically or via the config block (see here).

GUI

As of version `1.12.0.b0` of the firmware update tool, you can now provide a `.tar` or `.tar.gz` archive of the firmware you wish to upload to the board.

When you start the update tool you will see the following screen:



When you tick the `Flash from local file` option, an address bar will appear. Click the `...` button and locate the `.tar(.gz)` file with the firmware you wish to flash to your device. From this point the updater will behave just like a regular update but using the local file instead of downloading the latest.

Command line

You can also use the CLI version of the update tool to downgrade your device. Will need to get a `.tar` or `.tar.gz` archive of the firmware you wish to upload to the board. Then run the following commands:

```
$ pycom-fwtool-cli -v -p PORT flash -t /path/to/firmware/archive.tar.gz
```

Command Line Update Utility

Windows

After installing the Windows version of the updater tool, the CLI tool `pycom-fwtool-cli.exe` can be found here:

- 32-Bit Windows: `C:\Program Files\Pycom\Pycom Firmware Update\`
- 64-Bit Windows: `C:\Program Files (x86)\Pycom\Pycom Firmware Update\`

macOS

In order to get access to the CLI tool on macOS, you will need to right click on the Mac version of the updater tool and click `Show Package Contents`, then navigate to `Contents/Resources`, here you will find the `pycom-fwtool-cli`.

Linux

In the Ubuntu 14.04 LTS (and newer) version of the updater tool, `pycom-fwtool-cli` is installed in `/usr/local/bin`. In the Generic Linux package, the tool is extracted into folder `./pyupgrade`

Usage

```
usage: pycom-fwtool-cli [-h] [-v] [-d] [-q] [-p PORT] [-s SPEED] [-c] [-x]
                        [--ftdi] [--pic] [-r]
                        {list,chip_id,wmac,smac,sigfox,exit,flash,copy,write,write_rem
ote,wifi,pybytes,cb,nvs,ota,lpwan,erase_fs,erase_all}
                        ...

Update your Pycom device with the specified firmware image file For more
details please see https://docs.pycom.io/chapter/advance/cli.html

positional arguments:
  {list,chip_id,wmac,smac,sigfox,exit,flash,copy,write,write_remote,wifi,pybytes,cb,nv
s,ota,lpwan,erase_fs,erase_all}
    list                Get list of available COM ports
    chip_id             Show ESP32 chip_id
    wmac                Show WiFi MAC
    smac                Show LPWAN MAC
    sigfox              Show sigfox details
    exit                Exit firmware update mode
    flash               Write firmware image to flash
    copy                Read/Write flash memory partition
    write               Write to flash memory
    wifi                Get/Set default WIFI parameters
    pybytes             Read/Write pybytes configuration
    cb                  Read/Write config block
    nvs                 Read/Write non volatile storage
    ota                 Read/Write ota block
    lpwan               Get/Set LPWAN parameters [ EU868 US915 AS923 AU915]
    erase_fs            Erase flash file system area
    erase_all           Erase entire flash!

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         show verbose output from esptool
  -d, --debug           show debuggin output from fwtool
  -q, --quiet           suppress success messages
  -p PORT, --port PORT  the serial port to use
  -s SPEED, --speed SPEED
                        baudrate
  -c, --continuation    continue previous connection
  -x, --noexit          do not exit firmware update mode
  --ftdi                force running in ftdi mode
  --pic                 force running in pic mode
  -r, --reset           use Espressif reset mode
```

How to use the Parameters

The CLI tool uses a combination of global and command specific parameters. The **order of parameters** is **important** to avoid ambiguity.

`pycom-fwtool-cli [global parameters] [command] [command parameters]`

While `pycom-fwtool-cli -h` shows help for global parameters and a list of available commands, command specific parameters can be viewed using `pycom-fwtool-cli [command] -h`

The parameter `-r, --reset` has been added as a courtesy for users of 3rd party ESP32 products. This functionality is **not supported** by the Expansion Board 2.0 and may cause this tool to crash or hang in certain circumstances.

Global Parameters

```
`-h / --help`    : shows above help (you can also get detailed help for each sub-comma
nd
`-v / --verbose` : show verbose output from esptool.
`-d / --debug`   : show debug output from fwtool.
`-q / --quiet`   : suppress most output, used for scripting
`-p / --port`    : specifies the serial port to be used. Can also be set via **environ
ment variable ESPPORT**
`-s / --speed`   : specifies the serial speed to be used. Can also be set via **enviro
nment variable ESPBAUD**
`-c / --continuation` : continue previous connection in FTDI mode. This allows running
 multiple commands sequentially without having to reset the module. This option is ign
ored in PIC mode as the module can be reset via the serial connection.
`-x / --noexit`  : This will prevent the PIC from leaving firmware update mode.
`--ftdi`         : This will force the CLI updater to run in FTDI mode.
`--pic`          : This will force the CLI updater to run in PIC mode.
`-r, --reset`    : This will force the CLI updater to use Espressif's workaround to sw
itch into Firmware update mode. This reset method is intended for 3rd party hardware o
nly and is not supported by the Expansion Board 2.0
```

Commands

list

Get list of available serial ports ports.

```
usage: pycom-fwtool-cli list [-h]

optional arguments:
  -h, --help  show this help message and exit
```

Example: On macOS:

```
$ pycom-fwtool-cli  list
/dev/cu.usbmodemPy343431  [Pytrack] [USB VID:PID=04D8:F013 SER=Py343434 LOCATION=20-2]
/dev/cu.Bluetooth-Incoming-Port  [n/a] [n/a]
```

On Windows:

```
COM6  [Pytrack] [USB VID:PID=04D8:F013 SER=Py343434 LOCATION=20-2]
```

This is the only command that does not require any additional parameters.

All other commands require that **the serial port is specified either through the `-p` / `--port` option or through environment variable `ESPPORT`** You can optionally specify the speed either through `-s` / `--speed` or via environment variable `ESPBAUD`. The default speed is `921600`. The maximum speed for read operations on PIC based expansion boards & shields is `230400`. The speed will be reduced automatically if necessary.

Special note for Expansion Board 2.0

You will need to have a **jumper wire** connected between `G23` and `GND` to use any of the commands below. You will also need to **press the reset button** either before running each command or at least before running the first command. To avoid having to press the reset button again after each command, you can use the `-c` / `--continuation` option. The first command connecting to the device **MUST NOT** use the `-c` / `--continuation` option. This is to make sure a program called `_stub_` is uploaded onto the device. This `_stub_` cannot be uploaded more than once, so you need to tell the cli tool that the `_stub_` is already running, which is done through using the `-c` / `--continuation` option.

chip_id

Shows the unique ID of the ESP32 on the connected module.

```
usage: pycom-fwtool-cli  -p PORT exit [-h]

optional arguments:
  -h, --help  show this help message and exit
```

wmac

Shows the WiFi MAC of the connected module.

```
usage: pycom-fwtool-cli  -p PORT wmac [-h]

optional arguments:
  -h, --help  show this help message and exit
```

smac

Shows the LPWAN MAC of the connected module.

```
usage: pycom-fwtool-cli  -p PORT smac [-h]

optional arguments:
  -h, --help  show this help message and exit
```

sigfox

Show sigfox details

```
usage: pycom-fwtool-cli  -p PORT sigfox [-h]

optional arguments:
  -h, --help  show this help message and exit
```

exit

If a Pysense/Pytrack/Expansion 3 has previously been left in firmware update mode by using the `-x` option, this command can be used to exit the firmware update mode.

```
usage: pycom-fwtool-cli  -p PORT exit [-h]

optional arguments:
  -h, --help  show this help message and exit
```

flash

Writes firmware image to flash, must be as a `.tar(.gz)` file as provided by Pycom. These files can be found on GitHub.

```
usage: pycom-fwtool-cli  -p PORT flash [-h] [-t TAR]

optional arguments:
  -h, --help         show this help message and exit
  -t TAR, --tar TAR  perform the upgrade from a tar[.gz] file
```

copy

Read/Write flash memory partition from/to local file

```
usage: pycom-fwtool-cli  -p PORT [-h] [-p PARTITION] [-f FILE] [-r] [-b]

optional arguments:
  -h, --help            show this help message and exit
  -p PARTITION, --partition PARTITION
                        The partition to read/write (all, fs, nvs, factory,
                        secureboot, bootloader, partitions, otadata, fs1,
                        ota_0, config)
  -f FILE, --file FILE  name of the binary file (default: <wmac>-<part>.bin)
  -r, --restore         restore partition from binary file
  -b, --backup          backup partition to binary file (default)
```

write

Write to a specific location in flash memory.

```
usage: pycom-fwtool-cli  -p PORT write [-h] [-a ADDRESS] [--contents CONTENTS]

optional arguments:
  -h, --help            show this help message and exit
  -a ADDRESS, --address ADDRESS
                        address to write to
  --contents CONTENTS   contents of the memory to write (base64)
```

wifi

Get/Set default WiFi parameters.

```
usage: pycom-fwtool-cli wifi [-h] [--ssid SSID] [--pwd PWD] [--wob [WOB]]

optional arguments:
  -h, --help   show this help message and exit
  --ssid SSID  Set Wifi SSID
  --pwd PWD    Set Wifi PWD
  --wob [WOB]  Set Wifi on boot
```

pybytes

Read/Write pybytes configuration.

```
usage: pycom-fwtool-cli pybytes [-h] [--token TOKEN] [--mqtt MQTT] [--uid UID]
                                [--nwprefs NWPREFS] [--extraprefs EXTRAPREFS]

optional arguments:
  -h, --help            show this help message and exit
  --token TOKEN         Set Device Token
  --mqtt MQTT           Set mqttServiceAddress
  --uid UID             Set userId
  --nwprefs NWPREFS     Set network preferences
  --extraprefs EXTRAPREFS
                        Set extra preferences
```

Note: The local `pybytes_config.json` file is overwritten when making any modifications using this command (requires Pybytes firmware `1.17.5.b6` or higher and Firmware updater `1.14.3`).

cb

Read/Write config block (LPMAC, Sigfox PAC & ID, etc.). You can find the structure of this block here.

```
usage: pycom-fwtool-cli  -p PORT cb [-h] [-f FILE] [-b] [-r]

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  name of the backup file
  -b, --backup          backup cb partition to file
  -r, --restore         restore cb partition from file
```

If neither `-b` or `-r` is provided, the command will default to backup. If no file name is provided, `<WMAC>.cb` is used.

To backup your config block: `$pycom-fwtool-cli -p PORT cb`

To restore your config block: `$pycom-fwtool-cli -p PORT cb -r -f backup.cb`

nvs

Read/Write non-volatile storage.

```
usage: pycom-fwtool-cli  -p PORT nvs [-h] [-f FILE] [-b] [-r]

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  name of the backup file
  -b, --backup          backup cb partition to file
  -r, --restore         restore cb partition from file
```

If neither `-b` or `-r` is provided, the command will default to backup. If no file name is provided, `<WMAC>.nvs` is used.

To backup your NVS: `$pycom-fwtool-cli -p PORT nvs`

To restore your NVS: `$pycom-fwtool-cli -p PORT nvs -r -f backup.nvs`

ota

Read/Write ota block, this contains data relating to OTA updates such as the hash of the OTA firmware.

```
usage: pycom-fwtool-cli  ota [-h] [-f FILE] [-b] [-r]

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  name of the backup file
  -b, --backup          backup cb partition to file
  -r, --restore         restore cb partition from file
```

If neither `-b` nor `-r` is provided, the command will default to backup. If no file name is provided, `<WMAC>.ota` is used.

To backup your OTA block: `$pycom-fwtool-cli -p PORT ota`

To restore your OTA block: `$pycom-fwtool-cli -p PORT ota -r -f backup.ota`

lpwan

Get/Set LPWAN parameters saved to non-volatile storage. Please see here for more details.

```
usage: pycom-fwtool-cli  -p PORT lpwan [-h] [--region REGION]

optional arguments:
  -h, --help       show this help message and exit
  --region REGION  Set default LORA region
  --erase_region   Erase default LORA region
  --lora_region    Output only LORA region
```

erase_fs

Erase flash file system area. This is useful if some code running on the device is preventing access to the REPL.

```
usage: pycom-fwtool-cli  -p PORT erase_fs [-h]

optional arguments:
  -h, --help  show this help message and exit
```

erase_all

Erase entire flash, only use this if you are sure you know what you are doing. This will remove your devices lpwan mac addresses etc.

```
usage: pycom-fwtool-cli erase_all [-h]

optional arguments:
  -h, --help  show this help message and exit
```

Steps for using Secure Boot and Flash Encryption

Summary

In order to encrypt your firmware, you will need to build it from source. Our firmware source code can be found here, along with instructions on how to build it. Below you will find specific instructions on how generate keys, build and flash encrypted firmware.

1. Obtain keys (for Secure Boot and Flash Encryption)
2. Flash keys and parameters in `efuses`
3. Compile bootloader and application with `make SECURE=on`
4. Flash: bootloader-digest at address `0x0` and encrypted; all the others (partitions and application) encrypted, too.

Prerequisites

Firstly you will need to setup the tool chain and download the source code. detailed instructions on how to achieve this can be found here. Once you have complete this, you will need to open a terminal in the `esp32` folder of the firmware source code repo.

Next you will need keys for Flash Encryption and Secure Boot; they can be generated randomly with the following commands:

```
    python $IDF_PATH/components/esptool_py/esptool/espsecure.py generate_flash_encrypt
ion_key flash_encryption_key.bin
    python $IDF_PATH/components/esptool_py/esptool/espsecure.py generate_signing_key s
ecure_boot_signing_key.pem
```

The Secure Boot key `secure_boot_signing_key.pem` has to be transformed into `secure-bootloader-key.bin`, to be burnt into efuses. This can be done in 2 ways:

```
    python $IDF_PATH/components/esptool_py/esptool/espsecure.py extract_public_key --k
eyfile secure_boot_signing_key.pem signature_verification_key.bin
```

or, as an artifact of the make build process, on the same directory level as Makefile

```
    make BOARD=GPY SECURE=on TARGET=boot
```

To flash the keys (`flash_encryption_key.bin` and `secure-bootloader-key.bin`) into the efuses (write and read protected) run the following commands (ignoring the lines that start with `#`):

Note: Irreversible operations

```
    # Burning Encryption Key
    python $IDF_PATH/components/esptool_py/esptool/espefuse.py --port /dev/ttyUSB0 bur
n_key flash_encryption flash_encryption_key.bin
    # Burning Secure Boot Key
    python $IDF_PATH/components/esptool_py/esptool/espefuse.py --port /dev/ttyUSB0 bur
n_key secure_boot secure-bootloader-key.bin
    # Enabling Flash Encryption mechanism
    python $IDF_PATH/components/esptool_py/esptool/espefuse.py --port /dev/ttyUSB0 bur
n_efuse FLASH_CRYPT_CNT
    # Configuring Flash Encryption to use all address bits together with Encryption ke
y (max value 0x0F)
    python $IDF_PATH/components/esptool_py/esptool/espefuse.py --port /dev/ttyUSB0 bur
n_efuse FLASH_CRYPT_CONFIG 0x0F
    # Enabling Secure Boot mechanism
    python $IDF_PATH/components/esptool_py/esptool/espefuse.py --port /dev/ttyUSB0 bur
n_efuse ABS_DONE_0
```

If the keys are not written in efuse, before flashing the bootloader, then random keys will be generated by the ESP32, they can never be read nor re-written, so bootloader can never be updated. Even more, the application can be re-flashed (by USB) just 3 more times.

Makefile options:

```
    make BOARD=GPY SECURE=on SECURE_KEY=secure_boot_signing_key.pem ENCRYPT_KEY=flash_
encryption_key.bin TARGET=[boot|app]
```

- `SECURE=on` is the main flag; it's not optional
- if `SECURE=on` the following defaults are set:
  - encryption is enable
  - `secure_boot_signing_key.pem` is the secure boot key, located relatively to Makefile
  - `flash_encryption_key.bin` is the flash encryption key, located relatively to Makefile

For flashing the bootloader digest and the encrypted versions of all binaries:

```
    make BOARD=GPY SECURE=on flash
```

Flashing

For flashing the `bootloader-reflash-digest.bin` has to be written at address 0x0, instead of the `bootloader.bin` (at address `0x1000`).

Build is done using `SECURE=on` option; additionally, all the binaries are pre-encrypted.

```
make BOARD=GPY clean
make BOARD=GPY SECURE=on TARGET=boot
make BOARD=GPY SECURE=on TARGET=app
make BOARD=GPY SECURE=on flash
```

Manual flash command:

```
python $IDF_PATH/components/esptool_py/esptool/esptool.py --chip esp32 --port /dev/ttyUSB0 --baud 921600 --before no_reset --after no_reset write_flash -z --flash_mode dio --flash_freq 80m --flash_size detect 0x0 build/GPY/release/bootloader/bootloader-reflash-digest.bin_enc 0x8000 build/GPY/release/lib/partitions.bin_enc 0x10000 build/GPY/release/gpy.bin_enc_0x10000
```

OTA update

The OTA should be done using the pre-encrypted application image.

Because the encryption is done based on the physical flash address, there are 2 application binaries generated:

- `gpy.bin_enc_0x10000` which has to be written at default factory address: `0x10000`
- `gpy.bin_enc_0x1A0000` which has to be written at the `ota_0` partition address (`0x1A0000`)

Hint: on MicroPython interface, the method `pycom.ota_slot()` responds with the address of the next OTA partition available (either `0x10000` or `0x1A0000`).

MicroPython License Information