LoRa multipurpose device tutorials

1. Table of Contents

SICONIA
LoRa Alliance

2. Introduction

The Multi-purpose device is a compact field end-point, IP65, which is defined to help collecting physical events in different applications. It’s a generic and configurable device. Thanks to a java script you can create your own scenario using the embedded sensors.

The V1 version of the device is capable to collect information as:

- Temperature,
- 3 axes accelerations,
- Shock detection,

The V2 version will add the following features:

- Humidity,
- Barometer,
- Gyroscope.

The device is LoRa ™ certified and can be used on any LoRa Network Server able to use LoRa™ certified devices.

In this document, we will use the Sagemcom LoRa Netwkork Server as an example.

This document is applicable from firmware version 1.6

The latest API documentation can be accessed through the following URL:

http://193.253.237.167:380/loradevice-doc/javadoc.html

3. Tools presentation and installation

Used tools description

Javascript interpreter console

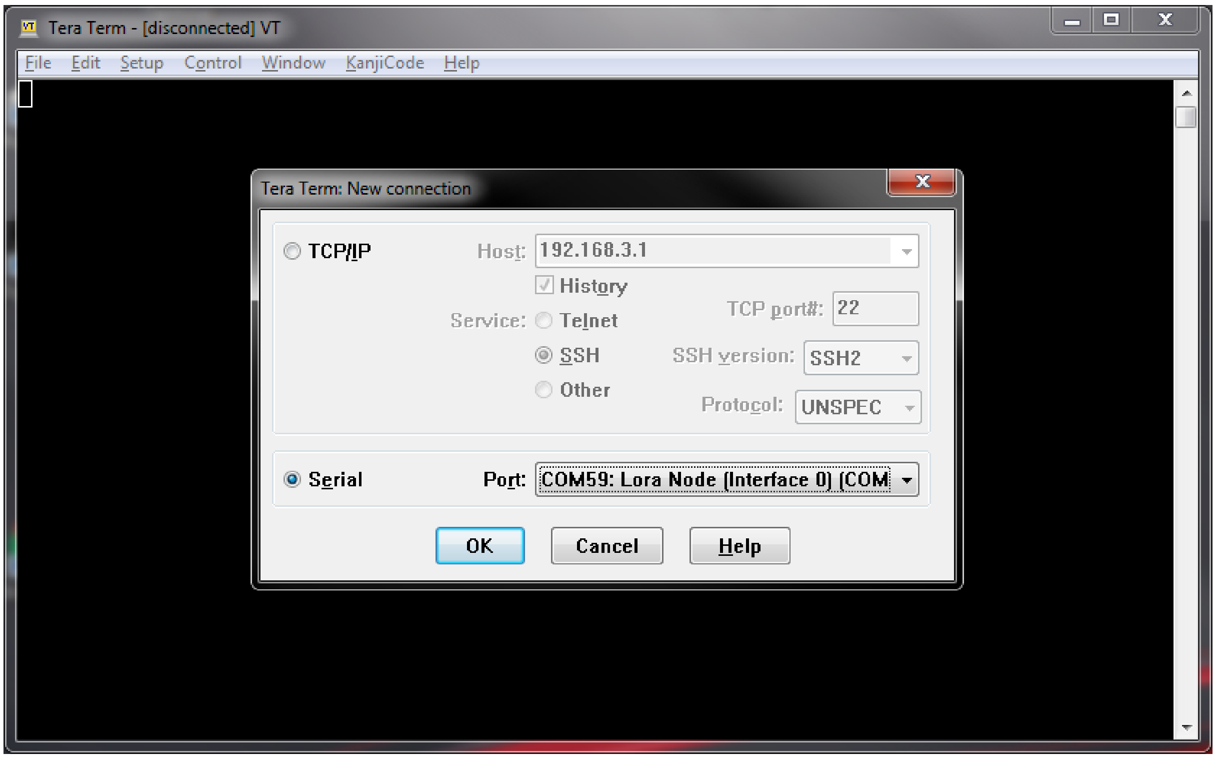
You will be able to connect to your device using a simple USB micro cable.

Once connected, you will need to install serial USB driver, which you can download from:

http://193.253.237.167:380/loradevice-doc/driver-pf01-vr02.zip

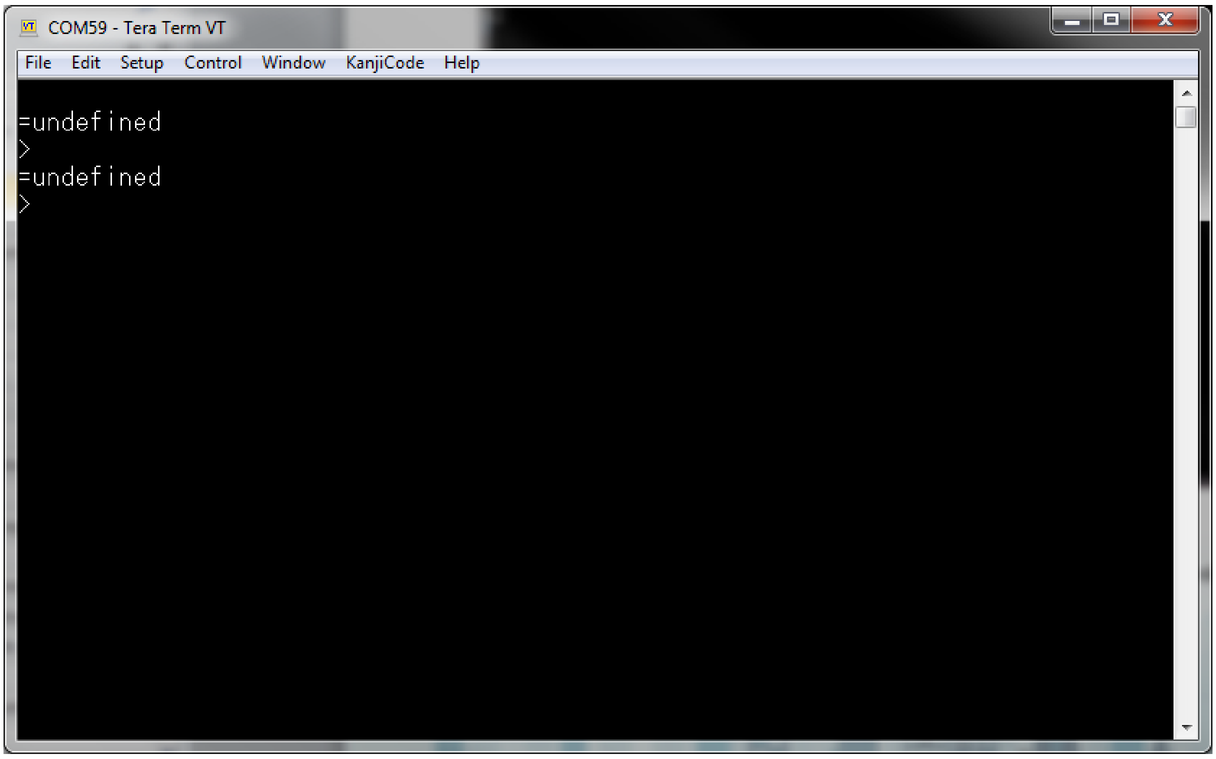
Note: These drivers are not signed. Therefore you will need to follow a special procedure under Windows™ 7 or superior to install them.

Once completed, you need to use your favorite serial console terminal application and open the LoRa node virtual COM Port.


Tera Term - [disconnected] VT
File Edit Setup Control Window KanjiCode Help
Tera Term: New connection
OK
Cancel
Help

TCP/IP
Host: 192.168.3.1
History [✓]
Service: Telnet [✓ SSH] Other
TCP port#: 22
SSH version: SSH2
Protocol: UNSPEC
[✓ Serial] Port: COM59: Lora Node (Interface 0) (COM

You should get a prompt, and pressing return should display an undefined result:



```
=undefined
>
=undefined
>
```

Depending of the used terminal application, you will need to change the parameter to get a clean result. For example, the interpreter sends only line feed at the end of a line.

In order to be able to copy/paste script into the console, it is better to add a millisecond delay between characters. Otherwise some character could be missed during the communication, and it will result to an unspecified behavior.

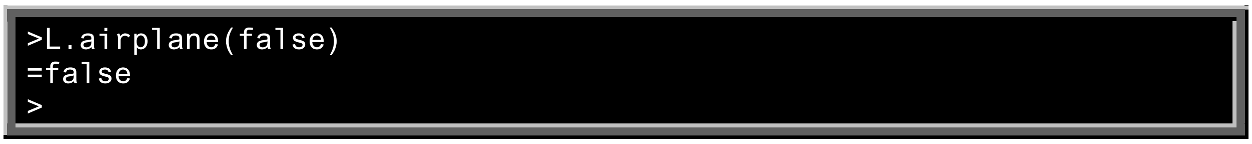
The provided console is a JavaScript interpreter, so it is able to store variable, calculate result. It gives you instant feedback then you can view and modify your program while it is running.

4. The “How to” send a HelloLoRaWorld

Airplane mode

First of all, we need to disable the airplane mode. When the device is shipped, it is positioned in airplane mode. This mode inhibits all radio operations, so LoRa frame will not be sent until we deactivate this mode.

In order to do this, the simple function “airplane” should be called, supplying in the arguments the new value of the mode. So we should start by:



```
>L.airplane(false)
=false
>
```

The device answers with the new value of the airplane mode (false in this case). The device is now able to communicate.

Duty cycle

If we want to send many frames to test, we also need to disable the LoRa duty cycle.

```
>L.dutycycle(false)
=false
>
```

The device also answer with the new value of the duty cycle (false indicate not activated).

Choosing ABP or OTAA

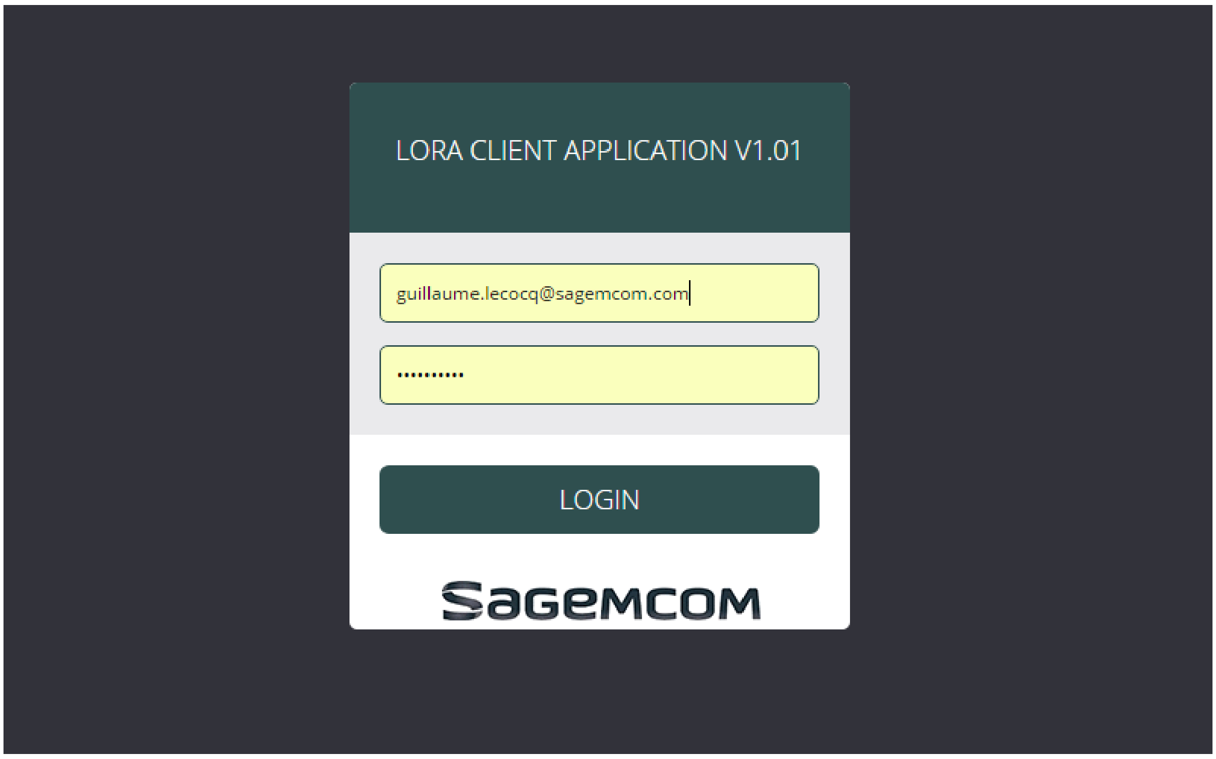
Depending of the needs, the device can operate in ABP or in OTAA mode, use the right chapter according to your needs.

Activation

As for any LoRa device, a preliminary declaration into your LoRa Network Server is needed. For this document we are using the Sagemcom's LoRa Network Server but of course you can use your own LNS.

Sagemcom LoRa client presentation

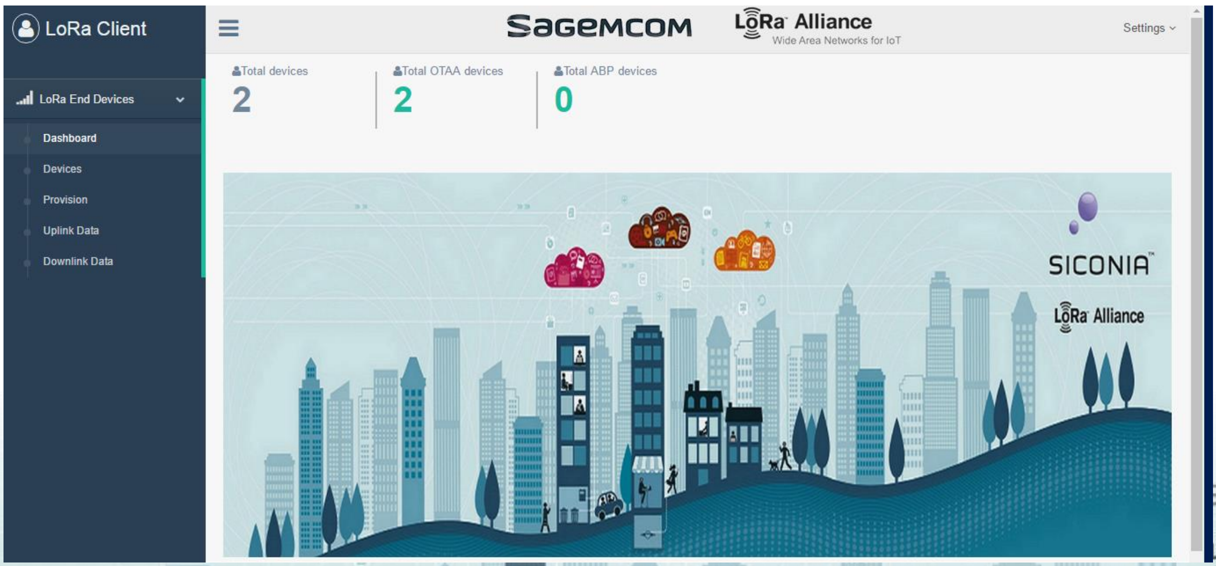
Use your login/password to login into the LoRa client:


LORA CLIENT APPLICATION V1.01

guillaume.lecocq@sagemcom.com

••••••••••

You should see the following dashboard page:



Total devices 2

Total OTAA devices 2

Total ABP devices 0

On the left, you can see the main menu:

- Dashboard : the current page
- Device list : the list of your current devices provisioned into the LoRa client
- Provision : the web page that allows you to create new devices
- Uplink data : to see the data coming from your devices
- Downlink data : a web page to allow you to send data to your devices

Activation By Personalization (ABP):

In order to add the device into the LoRa Network Server, you will need the following information:

- The device address [1]
- The AppSKey
- The NwkSKey

It is possible to retrieve the device address by typing the following:

```
>L.devAddr()
=[213, 103, 105, 1]
>
```

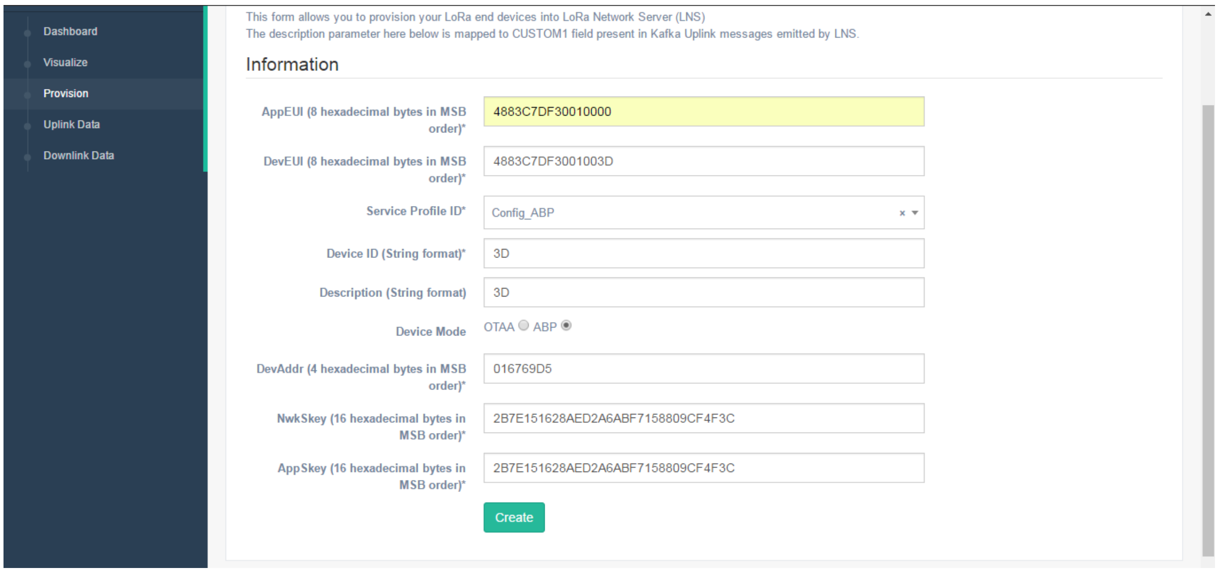
The AppSKey , the NwkSKey will be communicated upon request.

Note that the device address is displayed in little endian, and values are in decimal.

To add this device in ABP in your LoRa Network Server the entire field with the right information need to be provided.

For example, with the Sagemcom's LoRa client:

[1] Read chapter 0 to customize these values



This form allows you to provision your LoRa end devices into LoRa Network Server (LNS)
The description parameter here below is mapped to CUSTOM1 field present in Kafka Uplink messages emitted by LNS.

Information

AppEUI (8 hexadecimal bytes in MSB order)*	4883C7DF30010000
DevEUI (8 hexadecimal bytes in MSB order)*	4883C7DF3001003D
Service Profile ID*	Config_ABP
Device ID (String format)*	3D
Description (String format)	3D
Device Mode	OTAA ○ [✓ ABP ●]
DevAddr (4 hexadecimal bytes in MSB order)*	016769D5
NwkSkey (16 hexadecimal bytes in MSB order)*	2B7E151628AED2A6ABF7158809CF4F3C
AppSkey (16 hexadecimal bytes in MSB order)*	2B7E151628AED2A6ABF7158809CF4F3C

Over The Air Activation:

As for an ABP activation, in order to add your device into the LoRa network server, you will need the following information:

- The application unique identifier (AppEui)
- The device unique identifier (DevEui)
- The AppKey

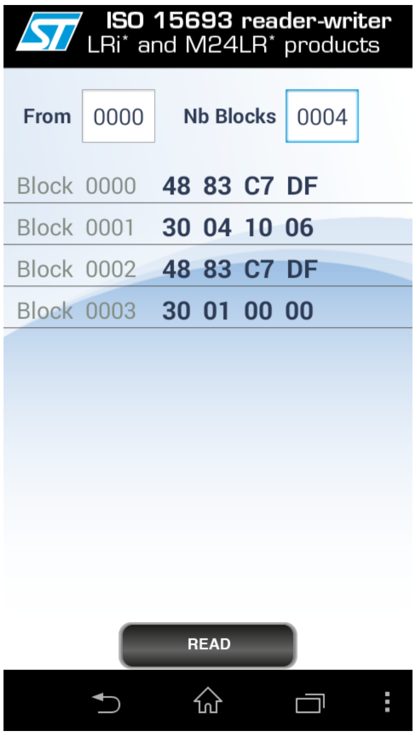
It is possible to retrieve the AppEui and DevEui directly from the device:

```
>L.devEui()
=[ 72, 131, 199, 223, 48, 1, 2, 61 ]
>L.appEui()
=[ 72, 131, 199, 223, 48, 1, 0, 0 ]
>
```

Values are displayed in **decimal format**. You may want to display them in hexadecimal format:

```
>var a = []
>L.devEui().forEach(function(e) { devEui.push(e.toString(16));});
=undefined
>devEui
=[ '48', '83', 'c7', 'df', '30', '1', '2', '3d' ]
>L.appEui().forEach(function(e) { appEui.push(e.toString(16));});
=undefined
>appEui
=[ '48', '83', 'c7', 'df', '30', '1', '0', '0' ]
>
```

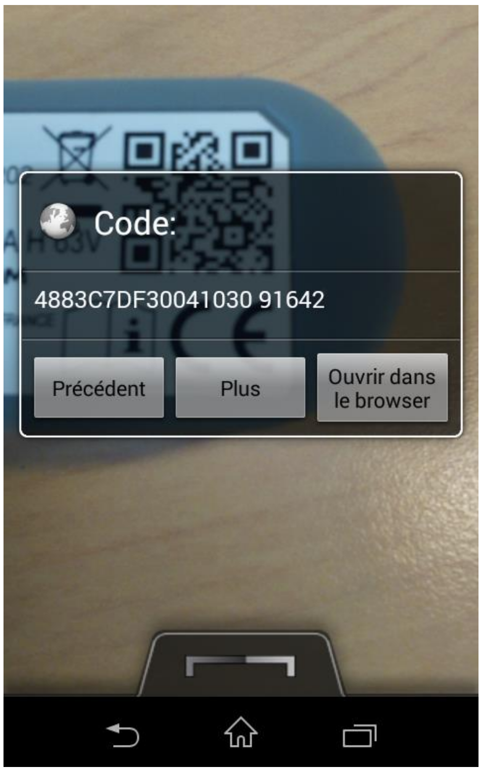
You can also use a NFC Reader if your phone is compatible. Then you can install a NFC Reader from ST Microelectronic. The DevEui is at position 0 in the NFC (8 bytes), and AppEui is at position 2 (8 bytes):


ISO 15693 reader-writer
LRi* and M24LR* products

From 0000 Nb Blocks 0004

```
Block 0000   48 83 C7 DF
Block 0001   30 04 10 06
Block 0002   48 83 C7 DF
Block 0003   30 01 00 00
```

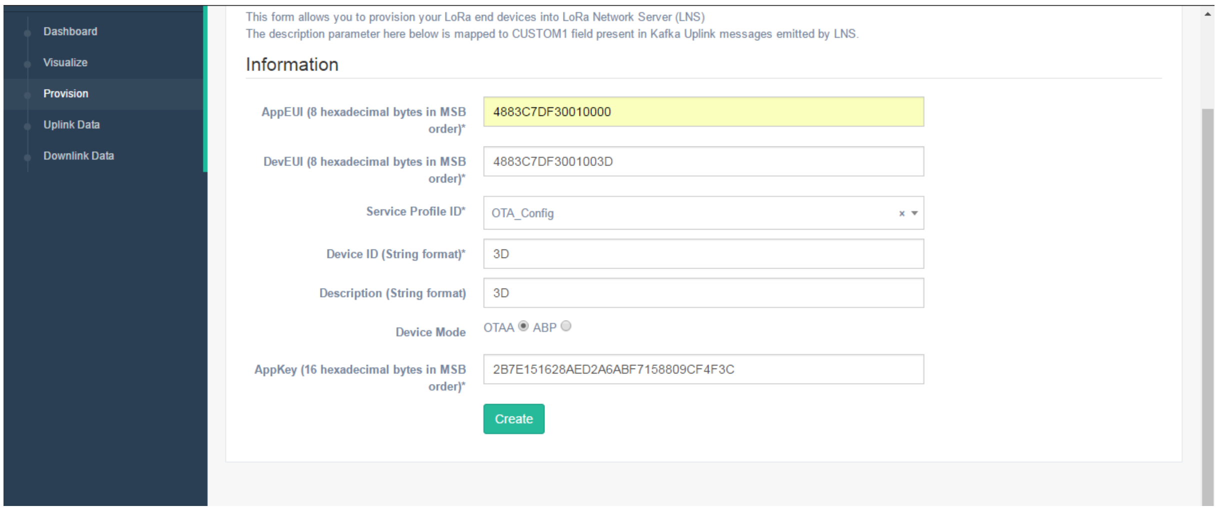
At last, you can also scan the QR Code present on the device.


Code:

4883C7DF30041030 91642

Note: the devEui ends before the space.

Once you have the information, you can fill up the Provision web page. For example with the Sagemcom’s LoRa client:



This form allows you to provision your LoRa end devices into LoRa Network Server (LNS)
The description parameter here below is mapped to CUSTOM1 field present in Kafka Uplink messages emitted by LNS.

Information

AppEUI (8 hexadecimal bytes in MSB order)*	4883C7DF30010000
DevEUI (8 hexadecimal bytes in MSB order)*	4883C7DF3001003D
Service Profile ID*	OTA_Config
Device ID (String format)*	3D
Description (String format)	3D
Device Mode	[✓ OTAA] ABP
AppKey (16 hexadecimal bytes in MSB order)*	2B7E151628AED2A6ABF7158809CF4F3C

Create

Note: you should use an OTAA configuration that is available for your user.

Note: The device ID helps you to identify this product. You can fill it with any information you want.

Join Request

When the device has been configured into the server, we need to ask the device to join the network:

```
>L.join()
=0
>
```

Shortly after, the device should send JoinRequest and if well configured into the server, it should answer with a JoinAccept message. We can check if the device has been accepted into the network:

```
>L.isJoined()
=true
>
```

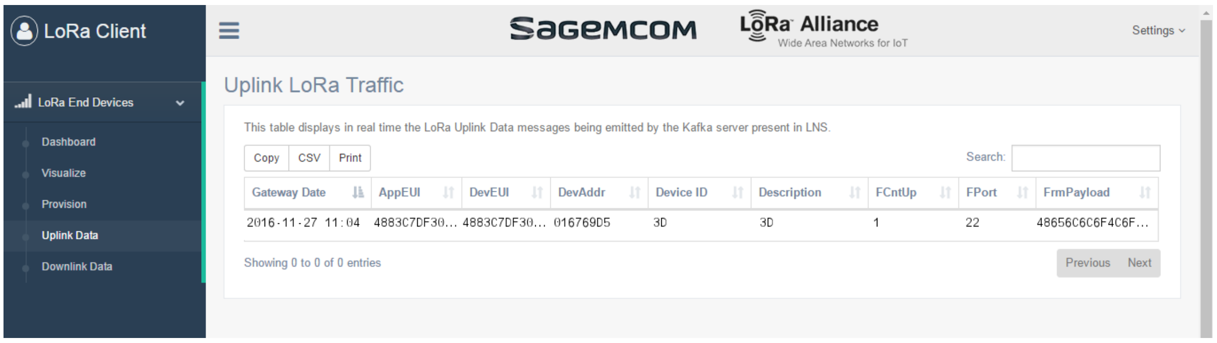
The true value indicates that the device has received a valid JoinAccept.

Sending HelloLoRaWorld

Now the device has been added into the network, it is time to send our first frame.

```
>L.send(22, “HelloLoRaWorld”)
=0
>
```

We just ask the LoRa stack to send a frame on FPort 22 containing the data “HelloLoRaWorld”. We should receive shortly the frame in the LoRa Client web interface:



Uplink LoRa Traffic

This table displays in real time the LoRa Uplink Data messages being emitted by the Kafka server present in LNS.

Gateway Date	AppEUI	DevEUI	DevAddr	Device ID	Description	FCntUp	FPort	FrmPayload
2016-11-27 11:04	4883C7DF30...	4883C7DF30...	016769D5	3D	3D	1	22	48656C6C6F4C6F...

Showing 0 to 0 of 0 entries

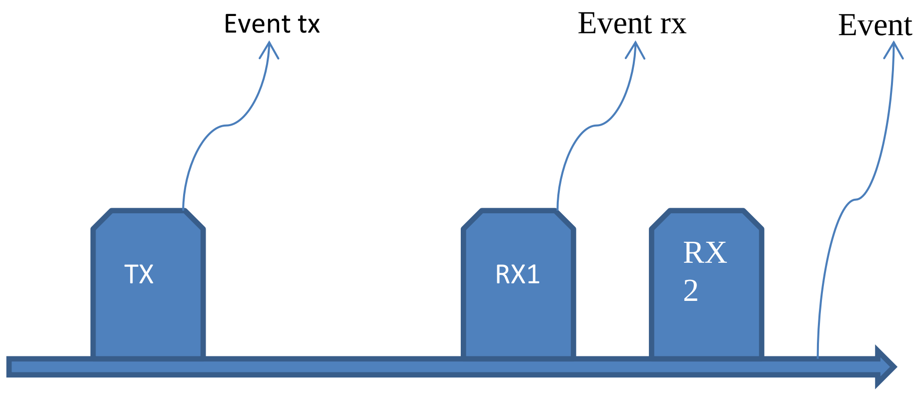
These data are string encoded, and are sent “as it” by the device. It is also possible to send the exactly same data as an array:

```
>L.send(22, [ 72, 101, 108, 108, 111, 76, 111, 82, 97, 87, 111, 114,
108, 100 ])
=0
>
```

5. Going further in LoRa communication

LoRa events : tx/rx/endtx/join

LoRa class provides a set of events which is interesting to enumerate. These events allows to register function that may be called when the events occurs. Multiple functions may be registered on the same event.



The ‘**tx**’ event is called when a frame has been sent.

The ‘**rx**’ event is called when a frame has been received.

The ‘**endtx**’ event is called once the complete transmission procedure (including an eventual reception) is finished.

The ‘**join**’ event is called when the device join the network.

Emission/Reception of LoRa messages

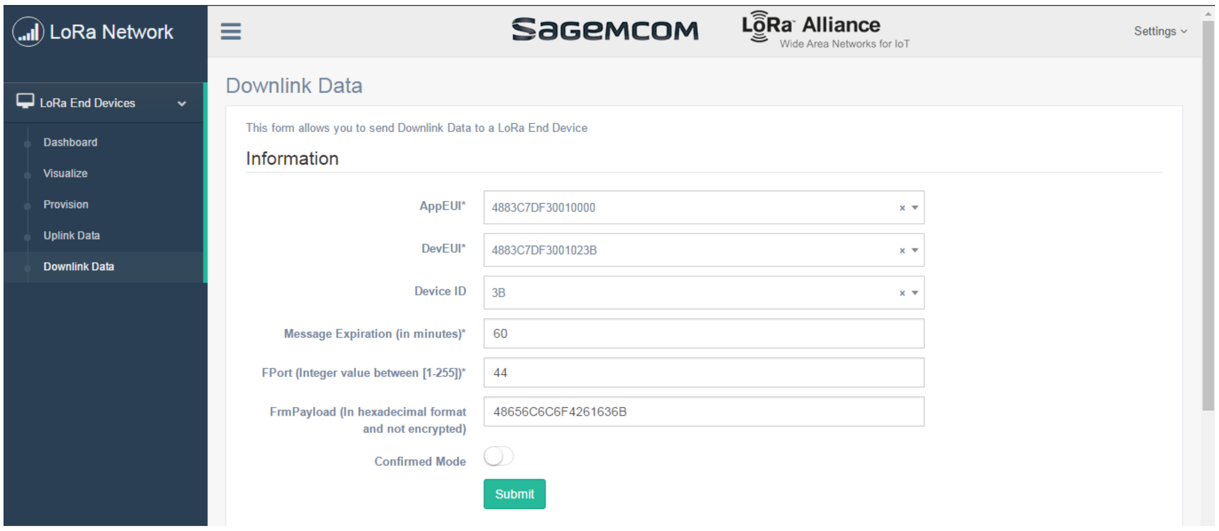
We will use the events in order to send and receive a LoRa message. Firstly, we will create the function that will be called when a message has been received:

```
function recv(status,port,data) {
    print("["+port+"] received : "+data);
}
```

Then we register this function on "rx" event:

```
>L.on('rx', recv);
=0
>
```

We now plan a downlink data using our LoRa client interface:



Downlink Data

This form allows you to send Downlink Data to a LoRa End Device

Information

AppEUI*	4883C7DF30010000
DevEUI*	4883C7DF3001023B
Device ID	3B
Message Expiration (in minutes)*	60
FPort (Integer value between [1-255])*	44
FrmPayload (In hexadecimal format and not encrypted)	48656C6C6F4261636B
Confirmed Mode	

Submit

And then we send a normal frame to open a reception slot:

```
>L.send(55, "Hi there!");
=0
>[44] received : [ 72, 101, 108, 108, 111, 66, 97, 99, 107 ]
```

Send confirmed messages

The device is also able to send confirmed message. In order to achieve this, two different functions are available:

- You can configure the stack in confirmed mode. Once configured, all message sent will require a confirmation from the server. For example, if you want to configure the stack in confirmed mode with 6 retries:

```
>L.confirm(8)
=0
>
```

- You can send a single message with confirmation from the server using the 'sendConf' method. The first parameter allows you to specify the number of retries:

```
>L.sendConf(3, 44, "MyMessage")
=0
>
```

It is also possible to register a function to be called once the message has been confirmed. First we create the function to be called:

```
function confirmFunction() {
    print("message confirmed");
}
```

Then we configure the stack in confirmed mode, adding the function as an argument:

```
>L.confirm(8, confirmFunction)
=0
>
```

LinkCheckRequest: estimate the link quality

It is possible to request for a link check using the function "lc".

First we create the function to be called:

```
function onLCAns(margin, nbGtw) {
    print("gtw : "+nbGtw);
}
```

Then we add the created function as the receiver of the event:

```
>L.on('lc', onLCAns)
=0
>
```

Last, we ask the device to send a LinkCheckRequest:

```
>L.lc(1)
=0
>gtw : 5
```

Explanation of payload manipulation

As seen in the previous chapter, data are sent "as it". Most of time, we want to 'encode' our data to be sent. JavaScript provides way to manipulate data: "arrays".

For example, if we want to send the temperature which is encoded in a simple int8, then send the long value.

6. Using LEDs Buttons to interact with user

Using leds to interact with user

There is a bicolor led on the device, and some function to use them. You can simply turn on/off the led:

```
>green.high()
=undefined
>green.low()
=undefined
>
```

Or you can ask the device to make them blink:

```
>red.blink(5)
=undefined
>orange.blink()
=undefined
>
```

The number in argument specifies how many the led will blink, without argument; the led will only blink once.

You may want to register the led to blink when a specific event occurs:

```
>L.on(‘tx’, ‘orange.blink()’)
=undefined
>
```

You can tune the delay between blink, for example if you want 2 seconds delay between each blink on led1:

```
>green.delay = 2000
=2000
>
```

Use buttons to let user interact with your device

A single button is present on the device, but there is many way to use it on the device.

At any time, you can get the status of the button. This will allow your application to know if the button is pressed or not:

```
>B.state()
=0
>
```

When we press on the button:

```
>B.state()
=1
>
```

There is special event that allows your application to catch short (<1 second) or long (>1 second) press:

```
>B.on('short', 'green.blink()')
=undefined
>B.on('long', 'red.blink()')
=undefined
>
```

There is also another event on button that allow application to know how many time a button has been pressed, to allow you to fully customize your device behavior. You need to create a function that will receive the event:

```
function onPress() {
    print("button pressed");
}

function onReleased(delayMs) {
    print("button released : "+delayMs+" ms");
}
```

And then register your function as an event receiver:

```
>B.on('press', onPressed);
=undefined
>B.on('release', onReleased)
=undefined
>button press
button released : 3467 ms
button pressed
button released : 265 ms
```

Please note that short and long event will still be called when the event release in registered.

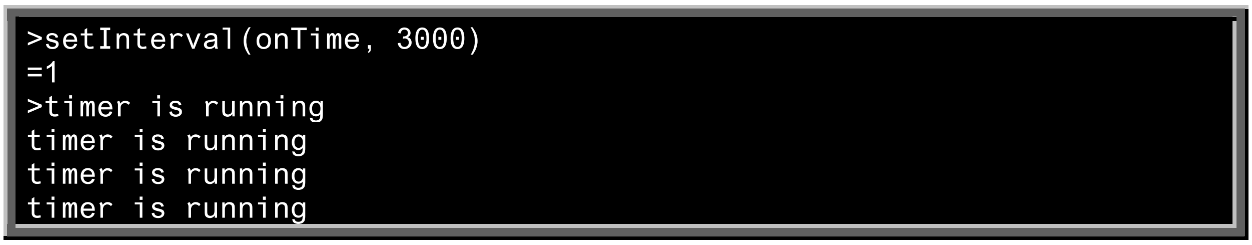
Interval/Timeout

Most of time, your application may want to have timing behavior. In order to achieve this, you can use setInterval to create a repeatable timer or setTimeout to create an oneshot timer.

When the timer ticks, it will launch the function specified on argument:

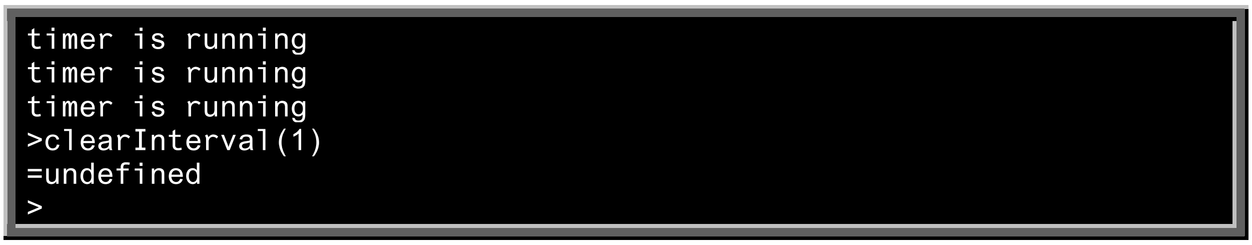
```
function onTime() {
    print("timer is running");
}
```

Then we create a repeatable timer:



```
>setInterval(onTime, 3000)
=1
>timer is running
timer is running
timer is running
timer is running
```

If we want to cancel it, we can use ‘clearInterval’ function:



```
timer is running
timer is running
timer is running
>clearInterval(1)
=undefined
>
```

7. Some usage of the accelerometer

The M Class

The accelerometer IC is handled by the built-in class M. User can configure the peripheral, get acquisitions and enable some event based functionalities. Please note that setting a high speed configuration may increase power consumption of the board.

Initialization, rate/frequency

The following method initializes the accelerometer with the default rate and scale which are 2g for scale and 1.6Hz for rate. This default configuration ensures the lowest power consumption of the sensor.

```
>M.init()
=0
>
```

It is possible to set an alternative configuration if needed. Below, the possible values for each parameter:

- Rate : [1.6, 12.5, 26, 52, 104, 208, 416, 833, 1660, 3330, 6660] expressed in Hz
- Scale : [2, 4, 8, 16] expressed in g

To set, the new configuration, call the initialization method with the chosen configuration

```
>M.init(2,104)
=0
>
```

If the accelerometer is not needed anymore, call the de-initialization method in order to shut down the peripheral and thus, reduce power consumption.

```
>M.deinit()
=undefined
>
```

Value acquisition

When the accelerometer is initialized, an acquisition can be obtained by calling the method above. The result is expressed in mg and represents respectively [x axis, y-axis, z-axis]

```
>M.read()
=[ -95, 18, 1020 ]
>
```

Motion detection

Motion detection is an event-based function. It can be used to detect a motion without consuming CPU time by polling continuously the accelerometer.

When enabling the functionality, we need to specify the minimum threshold value in mg and a callback that should be called when the threshold is reached on one or several axes.

```
function onMotion(axis) {
    print("Motion threshold reached in axis" + axis);
}
```

```
>M.motion(1000, onMotion);
=1000
>
```

Note: Due to hardware limitation, the real threshold registered in the accelerometer may be slightly different from the user specified threshold. The registered threshold is returned by the method. Please refer to "Sagemcom Software Reference" for more explanations.

Shock detection

Shock detection is an event-based function. It can be used to detect a shock signature without consuming CPU time by polling continuously the accelerometer.

When enabling the functionality, we need to specify the minimum threshold value in mg and a callback that should be called when the threshold is reached on one or several axes.

```
function onTap(axis) {
    print("Tap detected in axis" + axis);
}
```

```
>M.tap(1000, onTap);
=1000
>
```

Note: shock detection implies to have enough sampling frequency to be able to see the shock. The sampling frequency must be configured accordingly.

Using FIFO

The FIFO mode is also an event-based function. It can be used to collect an amount of samples at a precise rate without consuming CPU time.

First, we need to enable the FIFO mode with the chosen configuration in terms of rate and number of samples.

- Rate : [12.5, 26, 52, 104, 208, 416, 833, 1660, 3330, 6660] expressed in Hz
- Number of samples : [1, 680]. A sample format is [x axis, y-axis, z-axis]

```
>M.fset(104, 32);
=0
>
```

Then, we register a function to be called as soon as the FIFO is full.

```
function onFifoFull() {
    print(M.fget());
}
```

```
>M.on('fifo', onFifoFull);
=1000
>
[
[ -94, -91, -93, -94, -93, -94, -94, -93, -94, -92, -93, -90, -92, -
93, -95, -92, -92, -93, -94, -92, -91, -93, -92, -95, -92, -91, -96,
-94, -93, -94, -96, -93 ],
[ 24, 26, 24, 26, 23, 24, 26, 23, 27, 25, 25, 24, 22, 26, 25, 26,
24, 25, 25, 23, 23, 26, 25, 26, 25, 23, 23, 27, 24, 25, 24, 28 ],
[ 1018, 1023, 1012, 1024, 1016, 1027, 1016, 1016, 1018, 1024, 1019,
1020, 1019, 1023, 1020, 1019, 1016, 1022, 1020, 1023, 1018, 1017,
1020, 1021, 1022, 1020, 1017, 1023, 1018, 1021, 1024, 1021 ]
]
>
```

8. Temperature

The T Class

A standalone temperature IC is available for use. The class T in JavaScript handles the low level communication with this sensor.

Initialization

The following method initializes the temperature IC with the configuration that ensures the lowest power consumption.

```
>T.init()
=0
>
```

If the temperature sensor is not needed anymore, call the de-initialization method in order to shut down the peripheral.

```
>T.deinit()
=0
>
```

Value aquisition

When the temperature sensor is initialized, an acquisition can be obtained by calling the method above. The result is expressed in °C.

```
>T.read()
=25.68
>
```

Temperature monitoring

Instead of polling continuously the temperature value, it is possible to configure the sensor to raise an interruption when a temperature is reached. As soon as the interruption flag is raised, the MCU calls the event handler and if a user-defined function exists, it will be executed. Below, a prototype of the temperature callback:

```
>function onAlarm(high) {

     if (high == 1) {
           print("High limit reached\n")
     }
     else {
           print("Low limit reached")
     }
}
>T.on('temp', onAlarm);
```

In order to enable the monitoring, the following function should be called with the user defined parameters:

```
>var tHigh = 30;
=30
>var tLow = 10;
=10
>T.monitor(tHigh, tLow);
=0
>
```

Notes:

- If the user defined only “tHigh”, “tLow” will be automatically equal to “tHigh”.
- Temperature thresholds are floating numbers expressed in °C
- An event is executed only once as long as the temperature remains higher than the defined high threshold. The trigger is reactivated when the temperature falls below the low threshold.

9. Magnetometer

The Mg Class

The magnetometer IC is handled by the built-in class Mg. User can configure the peripheral and get raw or calibrated values.

Initialization

The following method initializes the magnetometer with the configuration that ensures the lowest power consumption of the sensor.

```
>Mg.init()
=0
>
```

If the accelerometer is not needed anymore, call the de-initialization method in order to shut down the peripheral and thus, reduce power consumption

```
>Mg.deinit()
=undefined
>
```

Value aquisition

When the magnetometer is initialized, a raw acquisition can be obtained by calling the method above. The result is expressed in mg and represents respectively [x-axis, y-axis, z-axis]

```
>Mg.readRaw()
= [ -264, -186, 145 ]
>
```

It is possible to get a calibrated acquisition by calling the following method:

```
>Mg.read()
=[ -388, -236, 88 ]
>
```

Calibration

By default, a factory calibration is used to compensate the board effect on the magnetometer sensor. It is possible to set a new calibration by calling the following method:

```
>var calibration = { 'offset' : [0, 0, 0 ], 'scale' : [1, 1, 1 ] }
"]
=Mg.calib(calibration)
=0
>
```

10. Humidity & Barometer

Humidity and Barometer APIs in JavaScript are slightly similar. They are respectively handled by ‘Hu’ and ‘Pr’ built-in static classes. Each class allows the user to configure and get acquisitions of each sensor. A calibration algorithm is implemented in the firmware in order to enhance the raw results.

Initialization

The following method initializes the Humidity sensor:

```
>Hu.init()
=0
>
```

Same for the Barometer:

```
>Pr.init()
=0
>
```

If the magnetometer is not needed anymore, call the de-initialization method in order to shut down the peripheral:

```
>Hu.deinit()
=undefined
>
```

Same method for the Barometer:

```
>Pr.deinit()
=0
>
```

Value aquisition

When the humidity sensor is initialized, a raw acquisition can be obtained by calling the method above. The result is expressed in %:

```
>Hu.readRaw()
=43.650
>
```

It is possible to get a calibrated acquisition by calling the following method:

```
>Hu.read()
=49.159999
>
```

Same methods are available for the barometer. However the result is expressed in hPa:

```
>Pr.readRaw()
=99360
>Pr.read()
=102173
>
```

11. Parameterize your application

Maybe you will want to be able to parameterize some values in your application. In order to achieve this, you could use the P class, which provide a simple dictionary to use in your application.

The dictionary work as this: you can set/get string values in the dictionary using a 4 bytes key (which could be a 4 characters string also).

It is possible to store any string values inside a parameter. If you want for example to store numerical value, you can also use parseInt or eval to retrieve the numerical value from a String.

```
>P.set(“FOO1”, “1000”)
=0
>P.get(“FOO1”)
=”1000”
>parseInt(P.get(“FOO1”))
=1000
>
```

The parameters are stored inside a non-volatile memory, and are flushed after a 3 seconds delay or after calling flush function.

```
>flush()
Saving all parameters in FLASH.....................
OK
=undefined
>
```

12. Controlling the lifecycle of your application

Reload

Your application code has been developed and you want it to be loaded at startup, you want to reset your application code from a precise point; the interpreter is able to save the current state inside the flash of the module.

```
>var myVar = 340
=340
>save()
=undefined
Erasing Flash.....................
Writing....
Compressed 38400 bytes to 1217
Checking...
Done !
>
```

Now if you reset the interpreter, it will reload the saved state.

```
>reset()
=undefined
Loading 1217 bytes from flash...
>myVar
=340
>
```

If you want to clean the saved state:

```
>clean()
=undefined
>
```

Init event

A special event 'init' is called when the interpreter starts. You are able to register function which will be at startup:

```
function onStart() {
    green.blink(3);
}
```

Then we can register this function to be called at init:

```
>board.on('init', onStart);
=undefined
>save()
=undefined
Erasing Flash.....................
Writing....
Compressed 38400 bytes to 1214
Checking...
Done !
>reset()
=undefined
Loading 1217 bytes from flash...
>
```

The green led should blink 3 times when reset is called.

You also are able at any moment to dump the current state of the interpreter:

```
>dump()
function onStart() {green.blink(3);}
Board.on("init", onStart);
=undefined
>
```

13. Some example of use

Make the LED blink when the device is joining the network

In this example, we will see how to:

- Make the orange led blink once when a message is sent without be register in the network
- Make the led become green when the device successfully join the network
- Make the red led blink 3 times when the join accept is received

First, we create and register our join event listener:

```
function onJoin() {
    red.blink(3);
}

L.on(‘join’, onJoin);
```

This listener will make the red led (led2) blink 3 times when the event is fired.

Create another event listener when a frame is sent:

```
function onTx() {
    if (L.isJoined()) {
        green.blink();
    } else {
        orange.blink();
    }
}
L.on(‘tx’, onTx);
```

In this function, we will check is the device is joined to the network. If it is, the green led will blink, otherwise, the orange led will blink.

At last, we create a timer that sends a frame regularly:

```
var t = setInterval(“L.send(11,’Hello’);”, 15000);
```

Start/Stop/Change scenario using the button

In this example, we will see how to use the button to start/stop a scenario using a press superior than 4 seconds.

The main idea in this example is the use of the function "eval", which tells the interpreter to interpret the content of a string, meaning that we are able to start function or code using a string.

So we will create two functions per scenario, formerly suffixed with "_start" and "_stop". When a scenario should be started, the method suffixed with "_start" would be called, and when it should be stopped, the method suffixed with "_stop" would be started.

So we register a "release" event listener:

```
var started = false;

function onRelease(d) {

    if (d>4000) {
        var suffix;

        if (started) {
            suffix = "_stop";
        } else {
            suffix = "_start";
        }

        eval("scenario1"+suffix+"();");
        started = !started;
    }

}
B.on('release', onRelease);
```

Now we will see how to evolve our script to allow the user to change between many scenarios when the device is not started.

The idea is to create and change a variable 'current' to contain the number of the scenario to start. When the user presses the button, the variable will be increased, and we will use an array to get the next scenario function name.

So we will change our script as follow:

```
var started = false;
var current = 0;
var allscenario = [ "scenario1", "scenarioFoo", "scenarioBar" ];

function onRelease(d) {

    if (d>4000) {
        var suffix;

        if (started) {
            suffix = "_stop";
        } else {
            suffix = "_start";
        }

        eval(allscenario[current]+suffix+"();");
        started = !started;
    } else if (d>1000 && !started) {
        current = (current+1) % allscenario.length;
    }

}
B.on('release', onRelease);
```

With the modification we made, the user will be able on 1 seconds press to increase the value of the variable current. When he will try to start again his device, it will be another scenario that will be started.

Send a frame on shock detection with configurable over LoRa parameter

In this example, we will demonstrate how to send a single frame on shock detection. In order to avoid sending a frame on each detection, we will add a simple timeout to avoid new detection before a configurable time.

First we create a function and we register it as the shock event:

```
function shock(axis) {
    L.send(17,axis);
}

M.init(2, 12.5);

M.tap(600, shock);
```

Now if a shock is detected, a frame will be sent. But we prefer to have a configurable timeout of blackout. So we modify our code as:

```
var blackoutTimer;
var blackout = false;

function shock(axis) {
    if (!blackout) {
        L.send(17,axis);
        blackoutTimer = createTimeout(“blackout=false”, 5000);
    }
}

M.init(2, 12.5);

M.tap(600, shock);
```

At last, we want the delay to be LoRa configurable. So we add our delay in a variable, and we create a function that allows changing it through LoRa:

```
var blackoutTimer;
var blackout = false;
var blackoutDelay = 5000;

function shock(axis) {
    if (!blackout) {
        L.send(17,axis);
        blackoutTimer = createTimeout("blackout=false", blackoutDelay);
    }
}

M.init(2, 12.5);

M.tap(600, shock);


function recv(s,p,d) {
    if (p == 55) {
        blackoutDelay = Utils.parseInt(d);
    }
}
L.on('rx', recv);
```

Now the device is able to receive LoRa frame on port 55 would allow changing the blackout delay.

14. FAQ

- How I prevent my device joining the network on each reset

The LoRa stack is not impacted by a JS interpreter reset. Therefore there is no need to call `L.join` on each init event. You can ask the joining status of the LoRa stack by using the '`L.isJoined()`' command. If the LoRa stack is still joined to the network, this method will return true, meaning you do not need to do a join request.

- How can I bypass the duty cycle

On development device, the duty cycle can be deactivated using the command '`L.dutycycle(false)`', and reactivated back using the command '`L.dutycycle(true)`'. You can also verify if the duty cycle is activated by calling the same method without arguments '`L.dutycycle()`'. This method has no effect on non-development device, and by default on startup the duty cycle is always activated (on development and non-development device).

- What is the difference between shock detection and motion detection

The motion function of the accelerometer allows getting an event when the acceleration of one of the axis is above a defined threshold. The advantage of such method is to delegate the monitoring to the accelerometer, is order to have the best consumption. This method is elementary, meaning that no further detection mechanism is implemented into the accelerometer after the detection.

The shock detection implemented by the accelerometer analyzes the signal waveform, and compare the duration and the maximum acceleration values to determinate if the acceleration is a shock or not. Three elements are important to considerate during shock detection: the threshold value, the activity duration and the silence duration. Therefore, a shock detection is only triggered when specific condition are met; an acceleration above the threshold value is detected and fall immediately before an activity delay (4 sampling period), and no acceleration is detected above the threshold value before a silence delay (2 sampling period).

- How can I determinate the remaining battery

The remaining battery can be retrieve directly by the application using the '`battery()`' function. This function returns a JavaScript array, which can also be label-accessed:

```
>Board.battery()
={ “capacity”: 850, “remaining”: 843, “percent”: 99 }
>Board.battery()[“percent”]
=99
>
```

- How much flash take a script? How can I reduce the memory usage of my scripts?

It is difficult to answer directly to this question. The JavaScript code cannot be directly estimated because it size depends of the code itself. But the remaining memory are accessible through the 'process.memory()' method:

```
>process.memory()
={ "free": 2291, "usage": 109, "total": 2400, "history": 19 }
>
```

- Does the security keys are all the same for all products? How can I retrieve the embedded keys?

No. The security keys are different per product, except for the development product which all always embedded the same keys.

There is no way to retrieve the security keys of a product, only Sagemcom is able to send you the key used inside your product.

- How can I check the firmware/hardware version embedded in my device?

There are two variables which contain the software and hardware version:

```
>swVersion
=260
>hwVersion
=514
>
```

- How can I maximize the battery life of my development device?

On development device, you have to:

- Reduce usage of the radio. The maximum consumption is taken from the LoRa radio. So the main idea to reduce the consumption is to develop your scenario without actually sending data over LoRa network, until it is necessary.
- Use the USB when possible. USB helps by providing energy to the device.
- Reduce the transmit power to the minimum required.

- My device keep disconnecting from USB. What is happening?

This is a normal behavior. The usb go to sleep after 5 minutes if it hasn’t receive any command from the PC. Use the usbSleep() method to deactivate this feature.