LAB ASSIGNMENT Nº2

LORAWAN-BASED ULTRASONIC DETECTOR

(DRAGINO LORA/GPS SHIELD, ARDUINO LEONARDO, THE THINGS NETWORK, FIREBASE, ANDROID STUDIO, NODE.JS)

1 INTRODUCTION

This assignment will introduce the students to Low Power Wide Area Network (LPWAN) technologies in general and LoRaWAN technology in particular. It builds upon the Lab Assignment Nº 1 in terms of persistent data storage and user interface. In fact, the Firebase project and Arduino app will require no further changes since the LoRaWAN system will seamlessly integrate with Firebase.

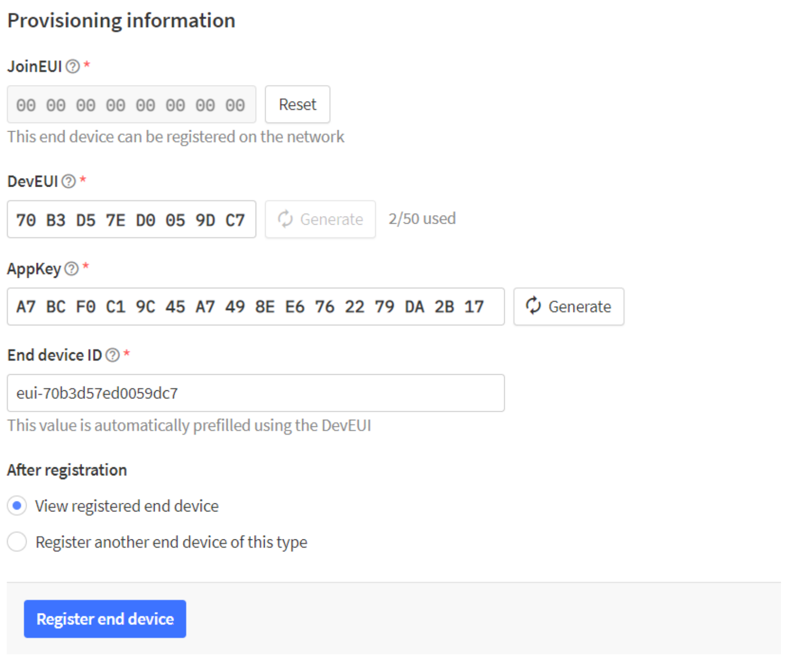
The instructions in this assignment are abridged so that, at each step, the students will have to investigate how to implement the requested functionality. At the end of this document, a list of useful references is provided.

2 CREATING AN ACCOUNT AND IOT APPLICATION IN THE THINGS NETWORK

This sequence steps will guide you through the creation of an account in The Things Network (TTN), within which an application will be registered. TTN is an open source LoRaWAN infrastructure that relies on its community of users to provide worldwide coverage. Users can register for free, create their applications and start to use the TTN infrastructure to receive data from their sensors at the TTN backend. There, the users can access the data through the Internet from any part of the World. They can also redirect that data to other servers. The users themselves may contribute installing their own LoRaWAN gateways, which are configured to connect to the router nodes and from them to the backend infrastructure. The security mechanisms of LoRaWAN allow the application data of user A to be sent to the backend through the gateway deployed by another user B, although user B is prevented from reading the data (though user B has access to the management information associated with the frames that carried the data from user A).

1) Go to the TTN website and create an account. If you already have an account, you can use it.
2) Go to the console and select the **Europe 1** (*eu1*, Dublin, Ireland) cluster. Enter the Applications area. Add a new application with the following profile:
    a. Owner: your user ID.
    b. Application ID: *rmic2223-group-XX*, where *XX* is the number of the group.
    c. Application name: you can use the Application ID.
    d. Description: *RMIC remote movement detection application*.

3) Once the application is created, it will be possible to add end devices. In order to be able to send/receive data to/from the backend, a device must be added. Click the option to add a device.
4) Select **Manually** for the type of registration of the LoRaWAN device, providing the requested parameters (see Figure 1):
    a. Frequency plan: Europe 863-870 MHz (SF9 for RX2 – recommended).
    b. LoRaWAN version: MAC V1.01.
    c. Regional parameters version: PHY V1.01.
    d. JoinEUI: 00 00 00 00 00.
    e. Dev EUI, AppEUI, AppKey: generate these values, which will have to be configured in the Arduino program (see below).
    f. End device ID: Some short character sequence to identify the device.
5) Once the device is configured, the application is ready to send/receive data to/from it. The next step is to configure and deploy the physical device.


Provisioning information

JoinEUI *
00 00 00 00 00 00 00 00
This end device can be registered on the network

DevEUI *
70 B3 D5 7E D0 05 9D C7
2/50 used

AppKey *
A7 BC F0 C1 9C 45 A7 49 8E E6 76 22 79 DA 2B 17

End device ID *
eui-70b3d57ed0059dc7
This value is automatically prefilled using the DevEUI

After registration
[✓ View registered end device]
Register another end device of this type

Figure 1: Entering device information in TTN.

3 SETTING-UP, DEPLOYING AND TESTING THE LORAWAN DEVICE AND APPLICATION

The device used in this assignment will perform the same tasks as the one in Lab Assignment Nº 1, but with a different MCU and communication technology. In fact, the sensor and actuator will be the same. The MCU will now consist of an Arduino Leonardo, whose pinout is depicted in Figure 2.

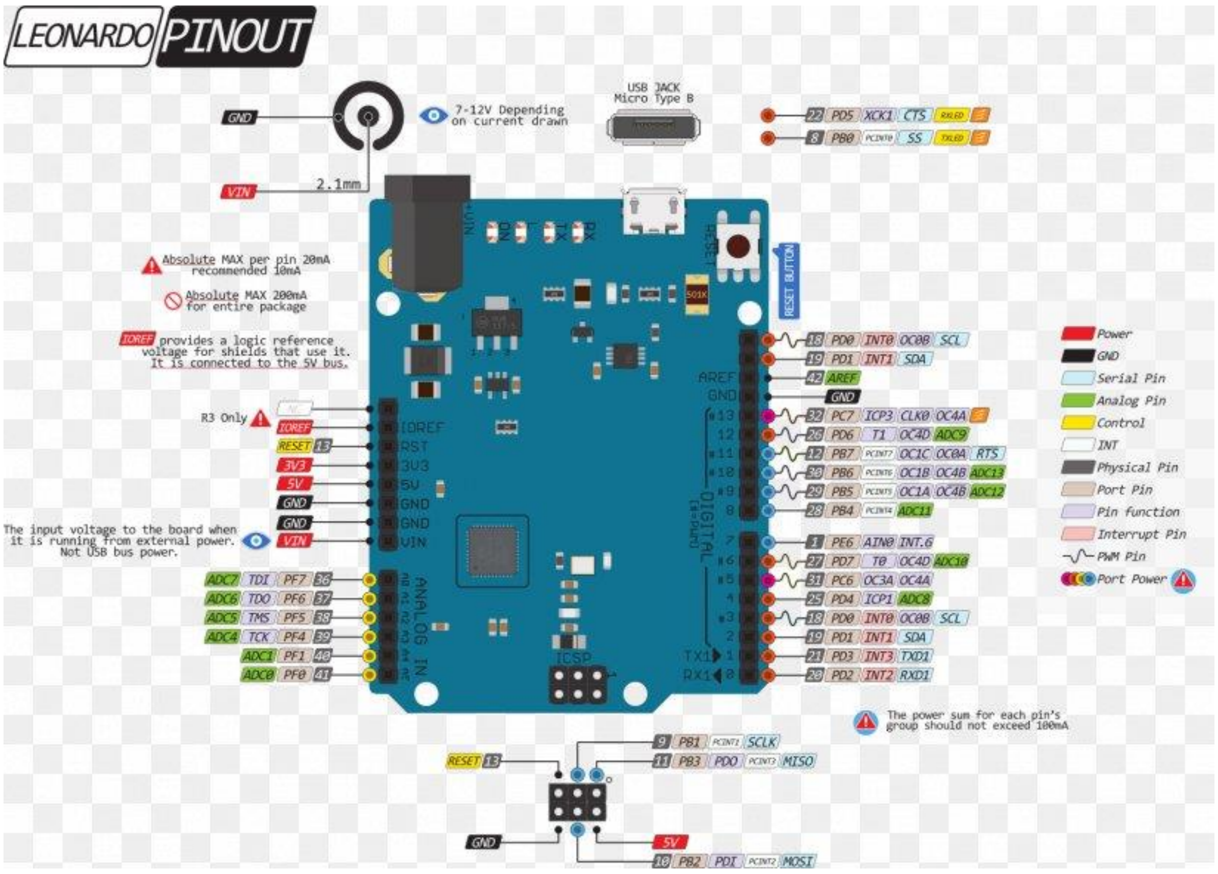

LEONARDO PINOUT
GND
7-12V Depending on current drawn
USB JACK Micro Type B
VIN
2.1mm
Absolute MAX per pin 20mA recommended 10mA
Absolute MAX 200mA for entire package
IOREF provides a logic reference voltage for shields that use it. It is connected to the 5V bus.
R3 Only
IOREF
RESET
3V3
5V
GND
VIN
The input voltage to the board when it is running from external power. Not USB bus power.
ANALOG IN
DIGITAL
ICSP
RESET BUTTON
Power
GND
Serial Pin
Analog Pin
Control
INT
Physical Pin
Port Pin
Pin function
Interrupt Pin
PWM Pin
Port Power
The power sum for each pin's group should not exceed 100mA
SCLK
MISO
MOSI
SCL
SDA
AREF

Figure 2: Pinout of Arduino Leonardo.

The LoRaWAN connectivity will be provided by the Dragino LoRa/GPS shield, whose pinout and **jumper configuration**[1] is depicted in Figure 3. The shield comes with a LoRa Bee module that implements the LoRa protocol. Although the shield implements a GPS receiver, we will only make use of the LoRaWAN functionality. Although the LoRa Bee module could be detached and used independently, the shield provides allows a more straightforward interface between the Arduino Leonardo and the LoRa Bee. In fact, most of the pins of the shield will directly connect and keep the function of the underlying Arduino pins.

[1] The jumper configuration must be carefully checked.

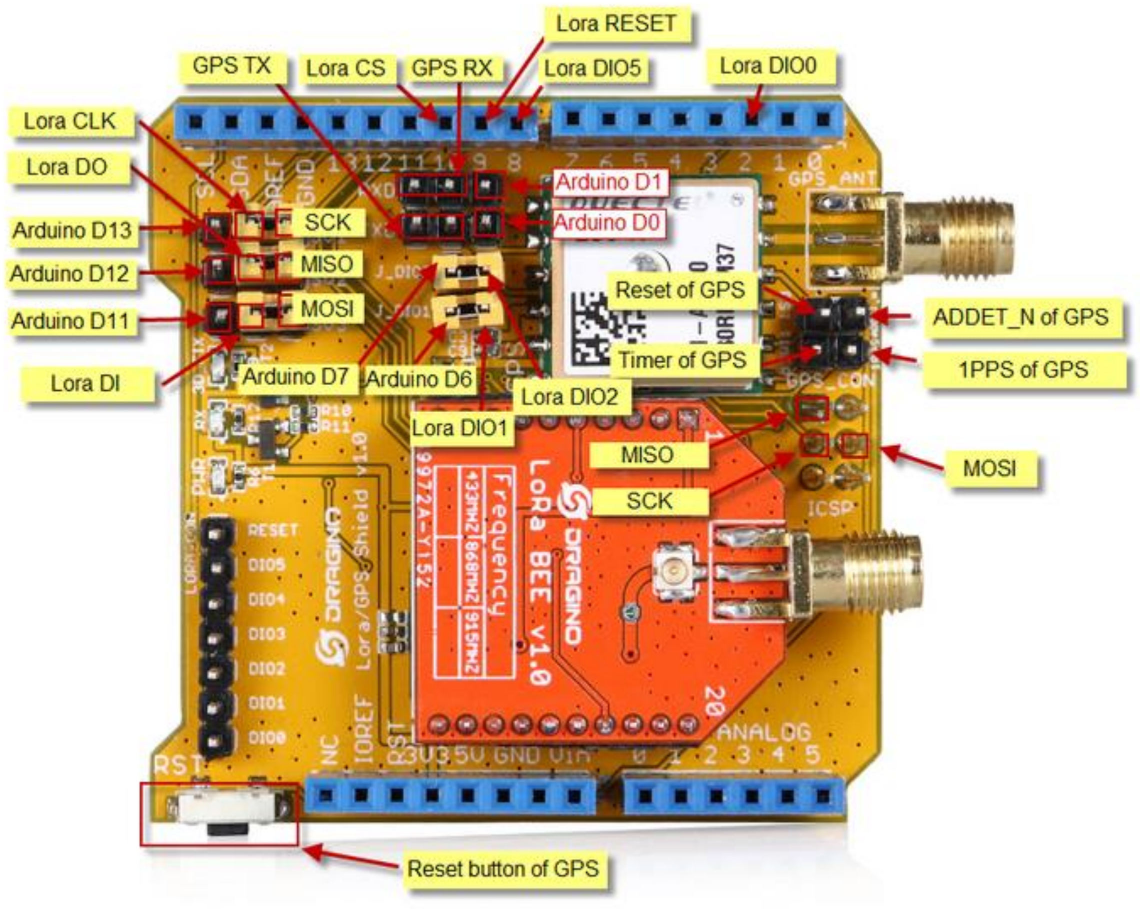


Figure 3: Dragino LoRa/GPS shield.

Arduino Leonardo will be fed by the PC through the USB interface. The LoRa/GPS shield will be attached on top of it and will connect with the HC-SR04 ultrasonic sensor, and the piezo buzzer, similarly to what was done in Lab Assignment Nº 1.

6) Mount the circuit as depicted in Figure 4 or Figure 4, depending on the used PIR sensor.

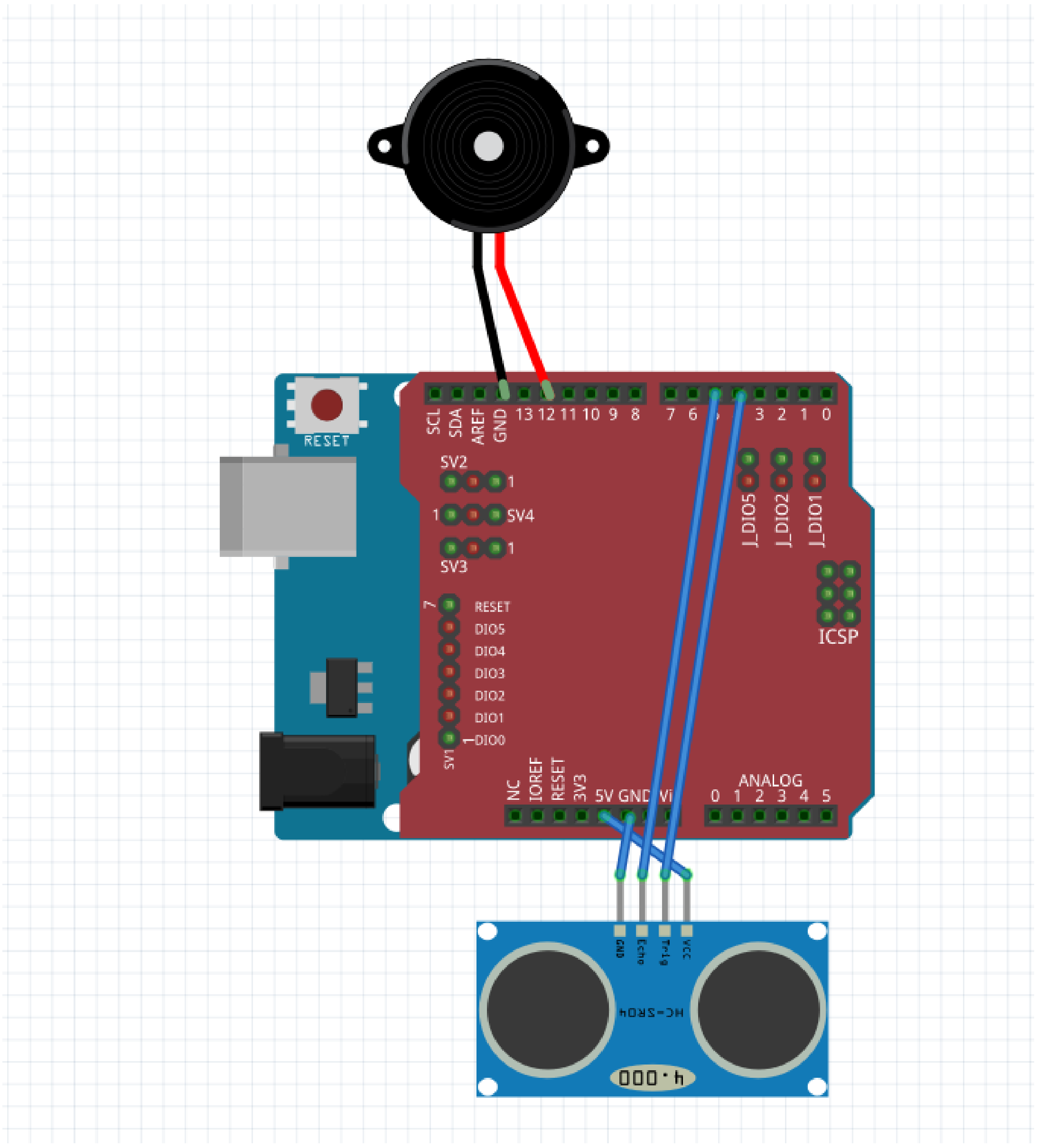
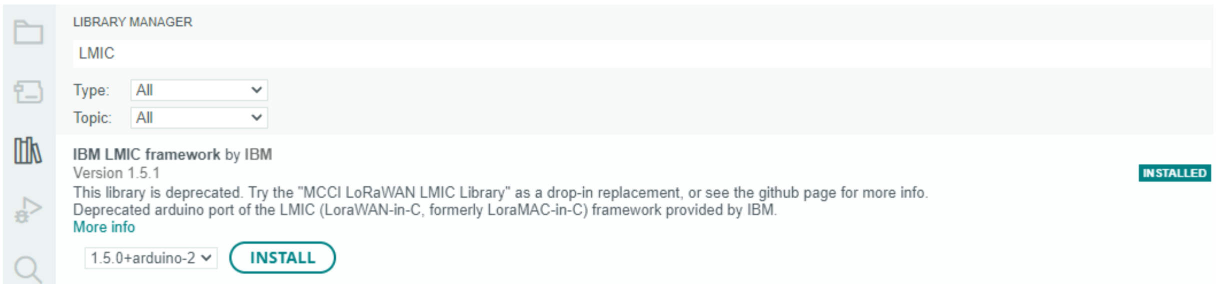


Figure 4: Ultrasonic detector and alarm circuit using Dragino GPS/LoRA shield on top of Arduino, and HC-SR04.

7) Install the IBM LMIC framework library as depicted in Figure 5.


LIBRARY MANAGER

LMIC

Type: All

Topic: All

IBM LMIC framework by IBM

Version 1.5.1

This library is deprecated. Try the "MCCI LoRaWAN LMIC Library" as a drop-in replacement, or see the github page for more info.
Deprecated arduino port of the LMIC (LoraWAN-in-C, formerly LoraMAC-in-C) framework provided by IBM.

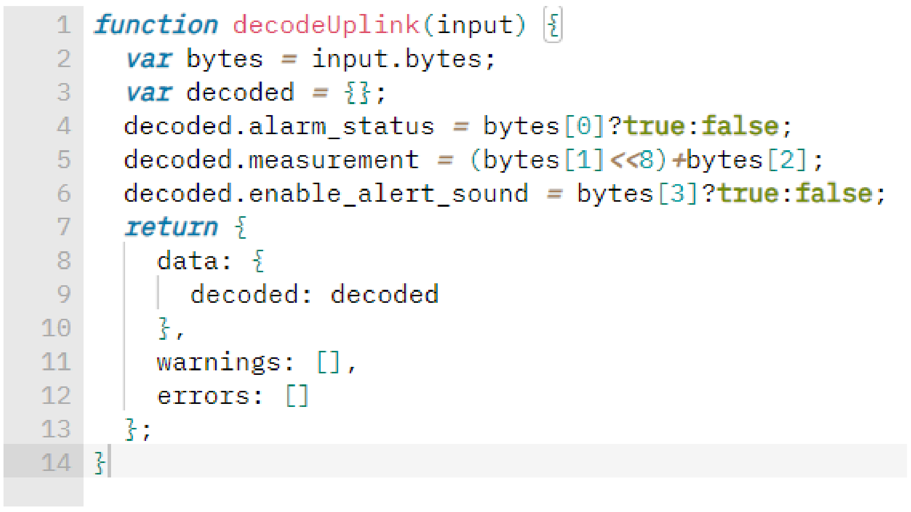
More info

1.5.0+arduino-2

Figure 5: Library Manager of the Arduino IDE with the IBM LMIC framework library installed.

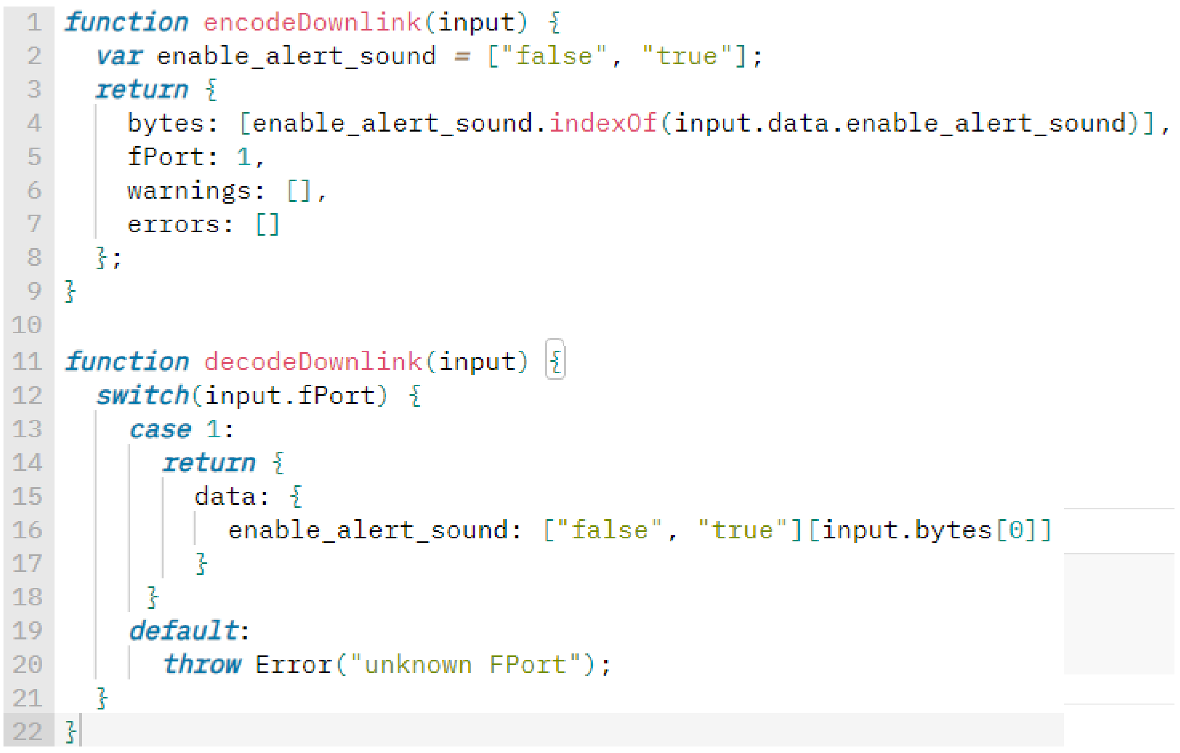
8) The program that will be run in Arduino Leonardo is given in file ***lorawanPIRSensor_v2.zip***. Configure the **Application EUI**, the **Device EUI**, and the **AppKey** with the values provided in the TTN device overview page. **Pay attention to the fact that these values must be filled in little-endian format.**
9) The program includes callback functions *check_sensor()* and *do_send()*, where the main actions are taken:
    a. The *do_send()* callback function will be called periodically in order to transmit the status of the device. The format of the payload is the following (each character represents one byte): MCCB, where M stands for the status of the PIR movement detection sensor (0x01 if motion is detected, 0x00 otherwise), CC stands for the alarm count (16 bits), and B stands for the buzzer enable variable configured in downlink data (0x01 if enabled, 0x00 otherwise).
    b. The *check_sensor()* callback function periodically checks the PIR movement detection sensor status and activates the Arduino Leonardo LED and buzzer accordingly (the latter will only be activated if the *buzzer_enable* variable has value 1). In case the sensor is currently detecting movement and the previous reading did not, the alarm count variable is incremented and the device tries to immediately transmit a packet.
    c. The LoRaWAN technology allows the transmission of a very limited amount of data in the downlink direction, which may be used to issue commands or configuration info to the device. In this application, one byte of data can be transmitted, which will set the value of the *buzzer_enable* variable. Due to the expected energy efficiency requirements of LoRaWAN, nodes can only receive downlink data in response to uplink frames (which denote that the device is active and not sleeping). As such, expect a delay between the instant in which downlink data is scheduled and the time when it is received. The procedure to send downlink data will addressed below.
10) It is useful to look at the debug messages sent to the serial interface, which can be visualized in the Serial Monitor of the Arduino IDE. Once the connectivity aspects are correct, you should be able to see the activation and data traffic in the TTN console, in the **Live data** separators of the application and of the device. You can check if the payload values are correct and match the expected device status.
11) The message payloads are reported in raw format. In order to read the payload data in a more user friendly way, you can set the Payload Format in the TTN application. The uplink payload format can be set in the **Payload formatters** separator of the application console. These can be set in Javascript for both uplink (device to application) and downlink (application to device). Program the payload formatters in Javascript, as shown in Figure 6 and Figure 7.
12) The uplink formatter defines and assigns the value of three subfields of the decoded object: *alarm_status* (Boolean), *measurement* (Integer) and *enable_alert_sound* (Boolean), whose meaning should now be familiar. Check that the decoder is well configured by looking at received uplink messages in **Live data**. The fields defined for the decoded object, as well as the respective values, should now appear reported in the logged messages.
13) The downlink formatter encoder allows downlink payloads to be specified and visualized in JSON format. The encoder receives a JSON object with a Boolean field named *enable_alert_sound*, transforming it into a one-byte payload of value 0x00 or 0x01,

respectively when it is *false* or *true*. The decoder assists displaying the decoded downlink message logs.



```
function decodeUplink(input) {
  var bytes = input.bytes;
  var decoded = {};
  decoded.alarm_status = bytes[0]?true:false;
  decoded.measurement = (bytes[1]<<8)+bytes[2];
  decoded.enable_alert_sound = bytes[3]?true:false;
  return {
    data: {
      decoded: decoded
    },
    warnings: [],
    errors: []
  };
}
```

Figure 6: Uplink formatter (decoder).

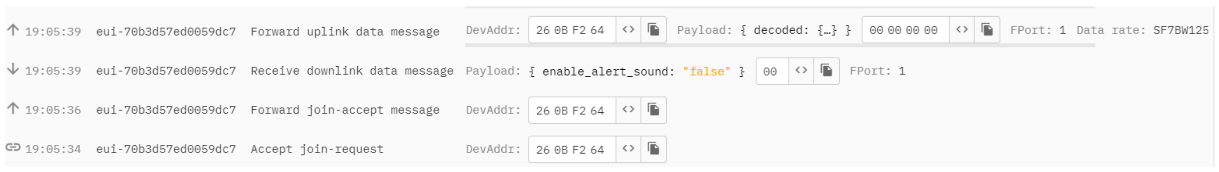


```
function encodeDownlink(input) {
  var enable_alert_sound = ["false", "true"];
  return {
    bytes: [enable_alert_sound.indexOf(input.data.enable_alert_sound)],
    fPort: 1,
    warnings: [],
    errors: []
  };
}

function decodeDownlink(input) {
  switch(input.fPort) {
    case 1:
      return {
        data: {
          enable_alert_sound: ["false", "true"][input.bytes[0]]
        }
      }
    default:
      throw Error("unknown FPort");
  }
}
```

Figure 7: Downlink formatter (encoder and decoder).

14) Downlink (i.e., from the application to the device) data can be scheduled directly from the TTN console. Access the device overview page and select the **Messaging** separator. Then, under the Downlink separator, schedule downlink payloads of one byte (0x00 or 0x01) to enable or disable the buzzer. Check that the **Live data** logs appear correctly and the device behaves as expected (see Figure 8). In the **Uplink** separator, uplink messages can also be created to test the uplink formatter.



```
↑ 19:05:39  eui-70b3d57ed0059dc7  Forward uplink data message    DevAddr: 26 0B F2 64  Payload: { decoded: {…} }  00 00 00 00  FPort: 1  Data rate: SF7BW125
↓ 19:05:39  eui-70b3d57ed0059dc7  Receive downlink data message  Payload: { enable_alert_sound: "false" }  00  FPort: 1
↑ 19:05:36  eui-70b3d57ed0059dc7  Forward join-accept message    DevAddr: 26 0B F2 64
  19:05:34  eui-70b3d57ed0059dc7  Accept join-request            DevAddr: 26 0B F2 64
```

Figure 8: Live data log in TTN.

4 INTEGRATING WITH A NODE.JS FIREBASE HTTP SERVER

By default, the TTN infrastructure will not provide persistent storage for the data. However, the uplink data can be sent to a database or to other systems if that is explicitly configured in the application by means of the Integrations mechanism. In this project, we will use an HTTP Webhook integration that will send the data to a Node.js webserver located at student's account in the **sigma.tecnico.ulisboa.pt** cluster. This webserver will receive HTTP requests from the TTN and reflect those requests on the Google Firebase database. On the other hand, it will receive event notifications from Google Firebase, which are transformed into downlink traffic HTTPS requests to the TTN. In order to obtain this functionality, take the following steps:

1) The Node.js version installed in **sigma.tecnico.ulisboa.pt** is by now outdated. As such we will install a more recent version locally (this assumes that you have still not done so). Locally install the most recent version of Node.js:
    a. Login at **sigma.tecnico.ulisboa.pt** using an SSH client, for example MobaXTerm.
    b. Execute "**cd Downloads**".
    c. Fetch the most recent version of Node.js. Assuming that it is version 14.18.0, execute "**wget https://nodejs.org/dist/v14.18.0/node-v14.18.0.tar.gz**".
    d. Execute "**tar –xvf node-v14.18.0.tar.gz**".
    e. Execute "**cd node-v14.18.0**".
    f. Execute "**./configure --prefix=$HOME/opt/node**".
    g. Execute "**make -j4 install**".
    h. You now have Node.js installed in *~/opt*. In order for the executables and libraries to be found by the shell, add the $PATH environment variable definition to *~/.bashrc*[2]: "**export PATH=~/opt/node/bin/:~/opt/node/lib/:$PATH**".
2) Node.js includes the Javascript package manager **npm** (Node Package Manager). You can now add the required Javascript packages, namely the one needed to interact with Google Firebase. Change directory to the Noje.js directory that was just created (*~/opt/node/*) and run the following command: "**npm install –g firebase-tools**".
3) Create the directory that will hold the webserver project, e.g.: "**mkdir ~/iot-alarm-app; cd ~/iot-alarm-app**".
4) Execute "**firebase login --no-localhost**". Copy/paste the login weblink into a browser and login to your Google Firebase account. Copy/paste the login code into the command line prompt. You are now logged-in.
5) Execute "**firebase init**". Select "Functions" from the menu: select with SPACE, then press Enter. Select "Use and existing project", then select the project implemented in Lab

[2] File *.bashrc* is located at the user's home directory and can be edited with any text editor, for example *nano*.

Assignment Nº 1. Select "Typescript", then "Yes" when prompted to have ESLint debug support. Respond "Yes" when prompted to install dependencies with **npm**.

6) Execute **"cd functions"**.
7) In file *tsconfig.json*, add the following compiler option: *""noImplicitAny": false"*. In file *package.json*, replace the *serve* script call definition with the following: *""serve": "npm run build && firebase serve -o* ***xxx.xxx.xxx.xxx*** *-p 55**XX** --only funct,"*, where ***xxx.xxx.xxx.xxx*** is the public IPv4 address of the sigma cluster computer[3], and ***XX*** is the number of the group (this is done to avoid port conflict in Sigma).
8) Replace the file ./src/index.ts with the one provided in the course's webpage.
9) Inside **~/iot-alarm-app/functions,** compile and run the webserver**: "npm run serve"**. Solve any bugs that are reported[4]. If the execution is correct, you will obtain an HTTP weblink for the Firebase function implemented by the webserver, which should be of the form: *http://xxx.xxx.xxx.xxx:55XX/iot-alarm-app/us-central1/helloWorld*. This will be user to configure the TTN integration, so that it knows where to sends POST requests whenever a new message from the device is received.
10) Go to the TTN console, access your application and go to separator **Integrations**, then **Webhooks**, and then a **Custom Webhook**. Set the webhook ID, and choose JSON for the webhook format. Configure the webhook URL so that it points to the webserver's weblink that was just indicated when running **"npm run serve"**. In the **Enabled messages** section, enable the uplink messages only, leaving the specific path blank. Add the webhook.
11) Go to separator **API Keys** and add an API key. This will authenticate webserver when issuing a downlink message to the device. After creating the API key, copy it and then go again to **Webhooks** and select the webhook just created. Paste the API key in the field **Downlink API key**.
12) By now, you should be able to see uplink messages reported in the **sigma** console. However, in order to allow downlink messages to be forwarded to your TTN application correctly, you must perform a small change to the */src/index.ts* script. In the **sigma** webserver console, wait for an uplink message and copy the URL in line **"REPLACE_PATH: https://eu1.cloud.thethings.network/api/v3/as/applications/xxxxxxxxxxx…",** and paste it into the */src/index.ts* script, line 118, in front of **"path:"**, replacing the default contents. After that, abort the webserver and restart it again with **"npm run serve"**.
13) Open your Firebase project and access the database. Both uplink and downlink integrations should be by now fully accomplished: updates from the device should be seen in the Firebase console, while changes to the *enable_alert_sound* variable made at the Firebase console should reach the device and change its behavior. The latter can also be monitored from the TTN console under **Live data**, which can also be used to identify some integration bugs.
14) Run the Android app developed in Lab Assignment Nº 1, and check that everything is running smoothly, with the app being able to interact with the LoRaWAN device.
15) **Optional:** Cloud Functions for Firebase is a serverless framework that lets you automatically run backend code in response to events triggered by Firebase features and HTTPS requests. The code that you have placed in the **sigma.tecnico.ulisboa.pt** cluster

[3] In order to obtain the latter, you may run the command **"ip addr"**.
[4] Each time one logs in and runs the emulator, the public IP address must be checked, since it may have changed in the meantime. This is a major cause for failure when running the emulator.

may also be deployed directly in Firebase. In order to do it, you need to create a billing account in Google Cloud and associate it with your project. **Creating a billing account entails providing a credit card number**. However, Google offers $300 in credit, which means that, in practice, during some time placement of the functions in Google Cloud will be free of charge. In order to place your TTN processing function in Google Cloud, you just have to execute the deploy command: **"firebase deploy --only functions"**.

5 REFERENCES

[1] Android developer guides, https://developer.android.com/guide.
[2] John Horton, "Android Programming for Beginners", Packt, https://www.packtpub.com/product/android-programming-for-beginners/9781785883262#:~:text=Android%20Programming%20for%20Beginners%20will%20be%20your%20companion,or%20are%20just%20looking%20to%20program%20for%20fun.
[3] Bill Phillips, Chris Stewart, Kristin, Marsicano, "Android Programming – The Big Nerd Ranch Guide", 3rd Edition, Big Nerd Ranch, 2017.
[4] TTN documentation, https://www.thethingsnetwork.org/docs/ .
[5] Node.js documentation, https://nodejs.org/docs/latest-v14.x/api/ .