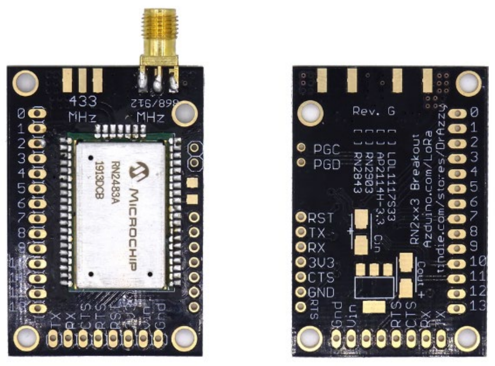
Arduino Pro Mini (ATmega328PB/ ATmega328P)
Part D: LoraWAN communication using RN2483

In this exercise / how-to-guide the Arduino Pro Mini will be connected with a RN2483 Lora/LoraWAN module which enables the device to communicate wirelessly over several kilometres **using a LoraWAN network**. Lora is a wireless communication technology that consumes considerably less power than traditional long-range technologies such as 2G, 3G and 4G. Background information on the LPWAN technology, Lora / LoraWAN, is essential for this guide (for example the LoraWAN lecture uploaded on DTU Learn).

Hardware requirements:

- Arduino Pro Mini with Arduino bootloader.
  TTL-USB adapter / RS232 used for UART communication with the board.
- RN2483 breakout board with 868 MHz antenna
- A sensor of some sort (temperature, humidity, or similar).
- Jumper wires (Dupont wires), breadboard etc. The usual stuff.



Rn2483 breakout board (front/back)

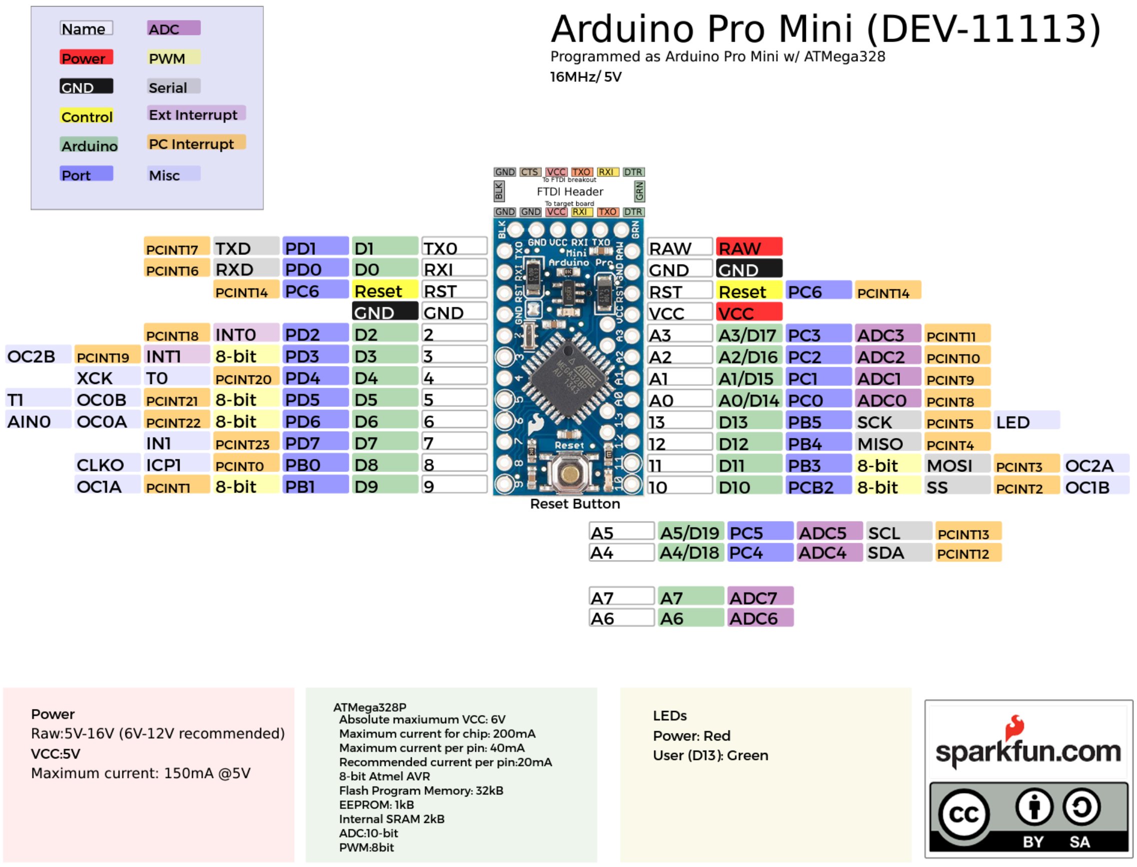
Software/document/other requirements:

- Arduino IDE https://www.arduino.cc/en/software
- Account on Helium LoraWAN network (https://console.helium.com)
- LoraWAN network coverage. See coverage map here: https://explorer.helium.com
- Microchip RN2384 datasheet:
  https://ww1.microchip.com/downloads/en/DeviceDoc/50002346C.pdf
- Microchip RN2384 command reference user's guide:
  https://ww1.microchip.com/downloads/en/DeviceDoc/RN2483-LoRa-Technology-Module-Command-Reference-User-Guide-DS40001784G.pdf

The **Qs**: Throughout the guide/exercise you will find questions (Q1, Q2..). Collect the answer in a document and submit this as a PDF on the DTU Learn assignment when done.
This exercise will be bundled with next week's exercise. I.e. deadline is at Oct 4th.

Arduino Pro Mini pinout

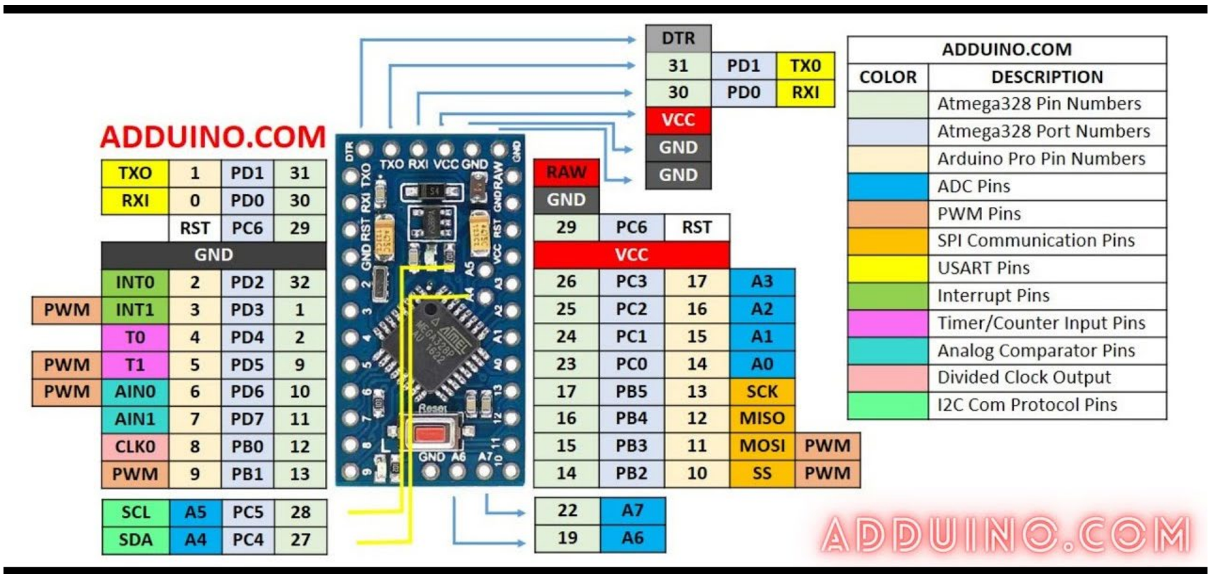
The Arduino Pro Mini comes in a few different board versions – depending on which company made the board. Below are two different board versions – identify which one you have and refer to that when you need to locate GPIOs or pins. Differences are mostly on the "short ends" of the board while the two "long sides" are often the same.


Arduino Pro Mini (DEV-11113)
Programmed as Arduino Pro Mini w/ ATMega328
16MHz/ 5V
Name
ADC
Power
PWM
GND
Serial
Control
Ext Interrupt
Arduino
PC Interrupt
Port
Misc
GND CTS VCC TXO RXI DTR
To FTDI breakout
FTDI Header
To target board
GND GND VCC RXI TXO DTR
BLK
GRN
PCINT17 TXD PD1 D1 TX0
PCINT16 RXD PD0 D0 RXI
PCINT14 PC6 Reset RST
GND GND
PCINT18 INT0 PD2 D2 2
OC2B PCINT19 INT1 8-bit PD3 D3 3
XCK T0 PCINT20 PD4 D4 4
T1 OC0B PCINT21 8-bit PD5 D5 5
AIN0 OC0A PCINT22 8-bit PD6 D6 6
IN1 PCINT23 PD7 D7 7
CLKO ICP1 PCINT0 PB0 D8 8
OC1A PCINT1 8-bit PB1 D9 9
RAW RAW
GND GND
RST Reset PC6 PCINT14
VCC VCC
A3 A3/D17 PC3 ADC3 PCINT11
A2 A2/D16 PC2 ADC2 PCINT10
A1 A1/D15 PC1 ADC1 PCINT9
A0 A0/D14 PC0 ADC0 PCINT8
13 D13 PB5 SCK PCINT5 LED
12 D12 PB4 MISO PCINT4
11 D11 PB3 8-bit MOSI PCINT3 OC2A
10 D10 PCB2 8-bit SS PCINT2 OC1B
Reset Button
A5 A5/D19 PC5 ADC5 SCL PCINT13
A4 A4/D18 PC4 ADC4 SDA PCINT12
A7 A7 ADC7
A6 A6 ADC6
sparkfun.com
BY SA

Power
Raw:5V-16V (6V-12V recommended)
VCC:5V
Maximum current: 150mA @5V

ATMega328P
Absolute maxiumum VCC: 6V
Maximum current for chip: 200mA
Maximum current per pin: 40mA
Recommended current per pin:20mA
8-bit Atmel AVR
Flash Program Memory: 32kB
EEPROM: 1kB
Internal SRAM 2kB
ADC:10-bit
PWM:8bit

LEDs
Power: Red
User (D13): Green



ADDUINO.COM	
COLOR	DESCRIPTION
🟩	Atmega328 Pin Numbers
🟦	Atmega328 Port Numbers
🟨	Arduino Pro Pin Numbers
🟦	ADC Pins
🟧	PWM Pins
🟧	SPI Communication Pins
🟨	USART Pins
🟩	Interrupt Pins
🟪	Timer/Counter Input Pins
🟦	Analog Comparator Pins
🟥	Divided Clock Output
🟩	I2C Com Protocol Pins

Setting up the Pro Mini and RN2483 module

1. Install the RN2483 Library: https://github.com/jpmeijers/RN2483-Arduino-Library in your Arduino IDE. If unsure how to install a library use instructions here: https://www.arduino.cc/en/Guide/Libraries

2. Make the following connections between your Pro Mini and RN2483 module. Its recommended to place both modules in a breadboard. Do not power the TTL or Pro Mini board yet.

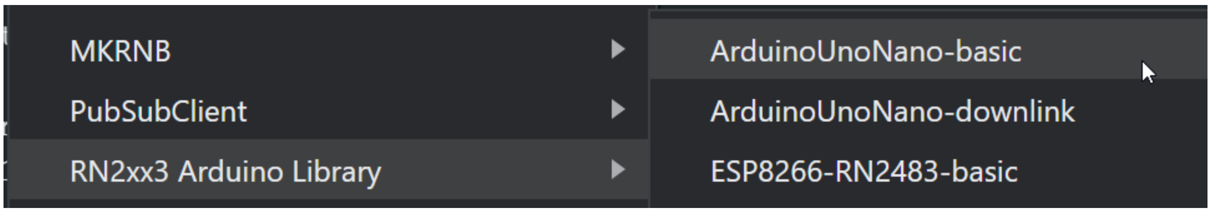
   Pro Mini **pin 11** <------> **RX** on RN2483
   Pro Mini **pin 10** <------> **TX** on RN2483
   Pro Mini **Vcc** <------> **3.3V** on RN2483
   Pro Mini **GND** <------> **GND** on RN2483
   Pro Mini **pin 12** <------> **RST** on RN2483

   Note: Confirm with the Pro Mini pinout if unsure which pins are where.

   Connect the antenna to the RN2483 board.

3. Connect the USB-TTL adapter to the Pro Mini board, and your computer. You should now have a connection like this:
   Laptop/Computer <-> TTL-USB adapter <-> Pro Mini <-> RN2483

4. In the Arduino IDE, load example code *Examples -> RN2xx3 Arduino Library -> ArduinoUnoNano-basic:*

   Note that this example code is made for joining and transmitting to a TTN network (The Things Network). We will use a different LoraWAN network. More on this later.

5. Inspect the code and analyse what it does. Change “TTN” to “Helium” in the code that so that meaningful debug messages are sent.

6. Now double check that you have connected the antenna and check that you have selected the correct board (Arduino Pro Mini) and correct Processor in Tools menu (8MHz or 16 MHz).
   Upload the code to the board and open the serial monitor. Remember to set the correct baud-rate. You will find this in the code in Arduino IDE.
   If you see an error message “*Communication with RN2xx3 unsuccessful. Power cycle*

the board." double-check the connections. Also try to push the RST button on the Pro Mini module or do a power-cycle by disconnecting and reconnecting the 3V3 supply pin on the RN2483 board.

7. If the setup and connection is successful, you should see something like this in the serial monitor (make sure you are using same baud rate in the serial monitor and RN2483 – check the code):

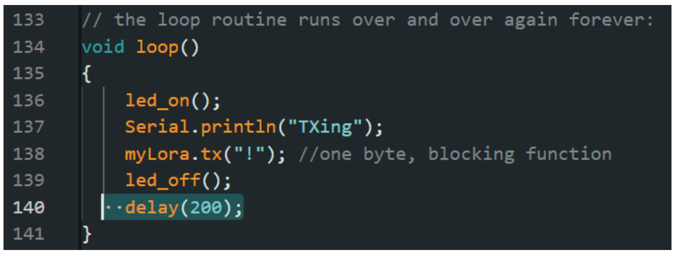
```
ct 31 2018 15:06:52
When using OTAA, register this DevEUI:
0004A30B00F2258B
RN2xx3 firmware version:
RN2483 1.0.5 Oct 31 2018 15:06:52
Trying to join TTN
Unable to join. Are your keys correct, and do you have TTN coverage?
```

From this output you will see the code and library have retrieved the hardware unique EUI number for the RN2483 module (DevEUI). This is a hardware identification number that we will need to add the device to the Helium LoraWAN server. The DevEUI was retrieved using this function:
  hweui = myLora.hweui();
Copy-paste the EUI number to a Notepad or similar. You will need it soon.

Note: You will also see a message that it was unable to join. This is because we have not inserted proper join credentials yet.

Q1: a) What is your DevEUI number for your RN2483 module? b) Is this a unique number or do all RN2483 modules have the same number? c) What format is this number (decimal, binary or hexadecimal)? What is your RN2483 firmware version?

8. The sketch code we use is made to do several things: 1. Retrieve the device EUI, 2. Attempt to join a LoraWAN network, 3. Transmit a test message "!" over and over again. In the default code, the "!" message is transmitted about 5 times per second (200ms delay) and this will breach national regulatory duty-cycle restrictions[1], as well as quickly deplete your free credits with Helium. Before continuing, edit the code to only send a packet every 10 seconds.

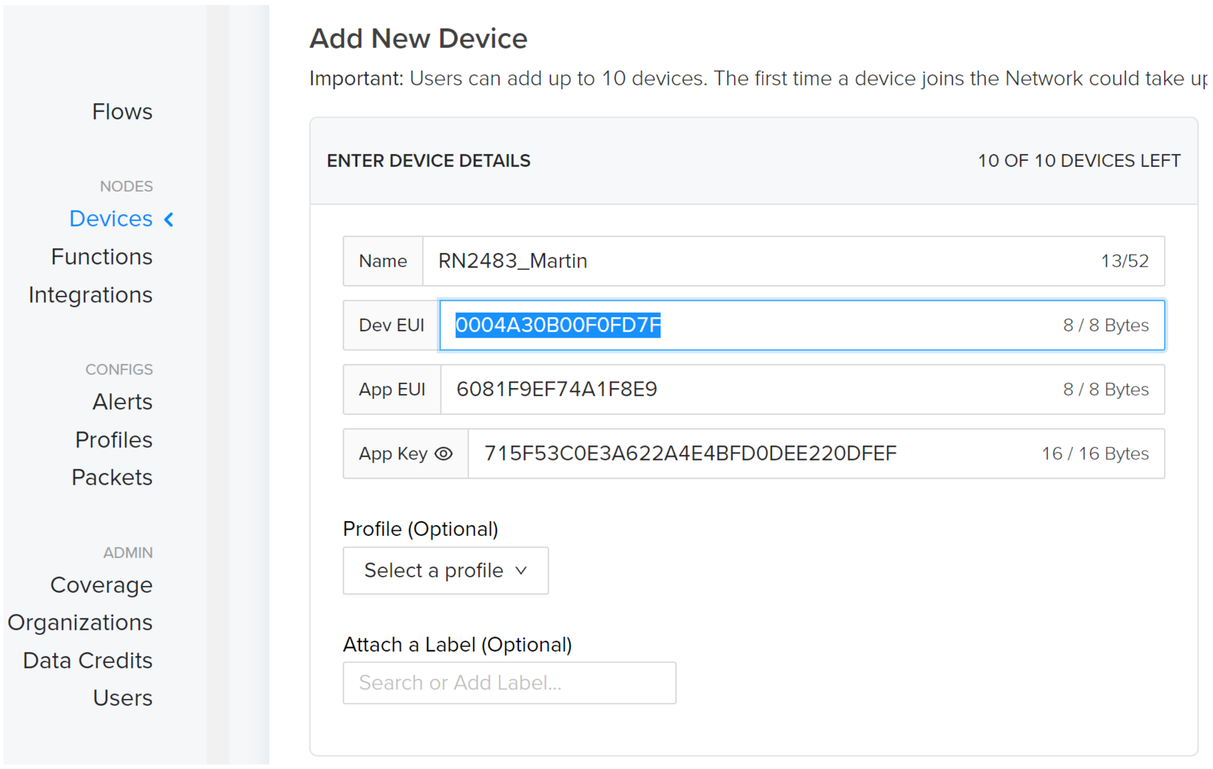


```
// the loop routine runs over and over again forever:
void loop()
{
    led_on();
    Serial.println("TXing");
    myLora.tx("!"); //one byte, blocking function
    led_off();
    delay(200);
}
```

[1] https://www.thethingsnetwork.org/docs/lorawan/duty-cycle/

Joining the LoraWAN network and transmitting data

9. Head over to https://console.helium.com and click Sign Up to register for an account.
10. Click Device and select to Add new device. Give it a name and insert the RN2483 device EUI you retrieved earlier via the serial monitor. Click Save when done.



Add New Device

Important: Users can add up to 10 devices. The first time a device joins the Network could take u

ENTER DEVICE DETAILS 10 OF 10 DEVICES LEFT

Name	RN2483_Martin	13/52
Dev EUI	0004A30B00F0FD7F	8 / 8 Bytes
App EUI	6081F9EF74A1F8E9	8 / 8 Bytes
App Key	715F53C0E3A622A4E4BFD0DEE220DFEF	16 / 16 Bytes

Profile (Optional)

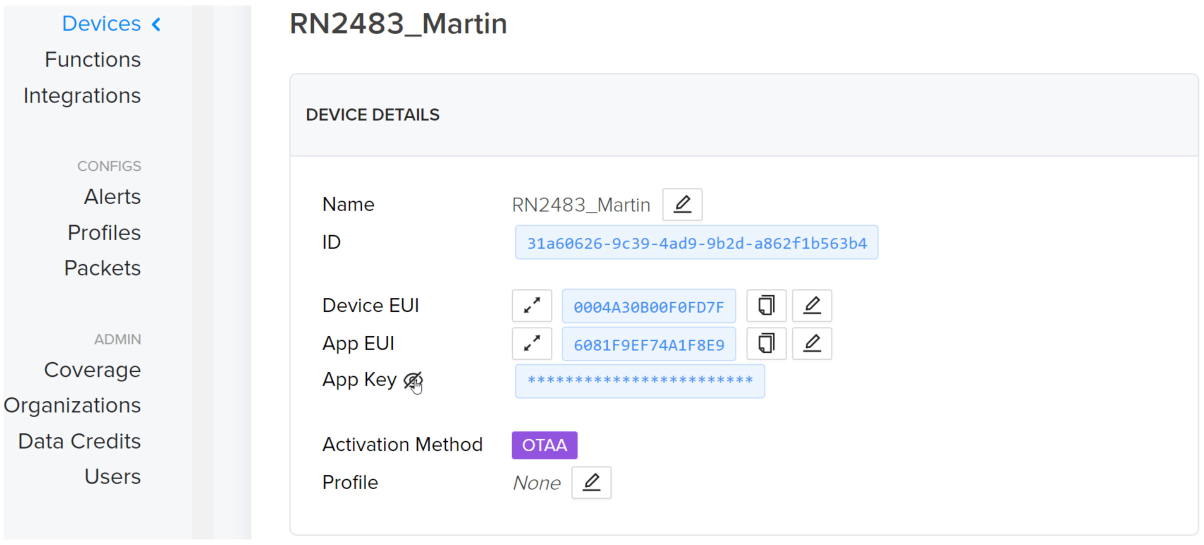
Select a profile

Attach a Label (Optional)

Search or Add Label...

11. Once saved, click the device again in the list. Now you will see something like below. Notice the activation method that is set to OTAA[2]. ABP is not available in Helium. Click the "eye" to reveal the *App key*. The *App EUI*, and the App Key, will be stored into the RN2483 module – this is done by the *myLora.initOTAA()* function in the Arduino example code. For more details, look in the *rn2xx3* library (rn2xx3.cpp file).

[2] OTAA: Over-The-Air Activation ABP: Activation By Personalization
https://www.thethingsindustries.com/docs/devices/abp-vs-otaa/

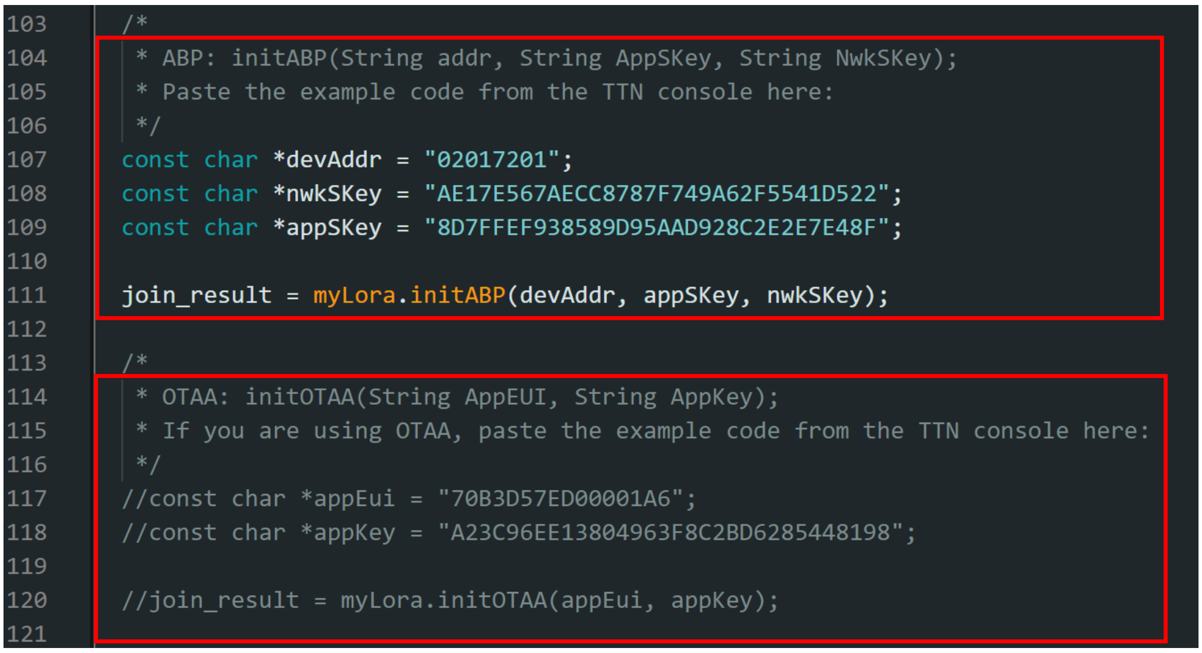


RN2483_Martin

DEVICE DETAILS

Name	RN2483_Martin
ID	31a60626-9c39-4ad9-9b2d-a862f1b563b4
Device EUI	0004A30B00F0FD7F
App EUI	6081F9EF74A1F8E9
App Key	************************
Activation Method	OTAA
Profile	None

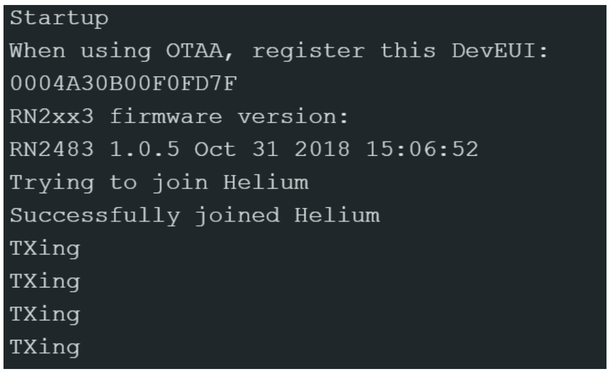
12. Now return to the Arduino IDE. In the code we have the option of using either OTAA or ABP as join network and activation method. You will need to un-comment the OTAA lines and insert the AppEUI and AppKey you got from the Helium server. Comment the ABP related lines.



```
/*
 * ABP: initABP(String addr, String AppSKey, String NwkSKey);
 * Paste the example code from the TTN console here:
 */
const char *devAddr = "02017201";
const char *nwkSKey = "AE17E567AECC8787F749A62F5541D522";
const char *appSKey = "8D7FFEF938589D95AAD928C2E2E7E48F";

join_result = myLora.initABP(devAddr, appSKey, nwkSKey);

/*
 * OTAA: initOTAA(String AppEUI, String AppKey);
 * If you are using OTAA, paste the example code from the TTN console here:
 */
//const char *appEui = "70B3D57ED00001A6";
//const char *appKey = "A23C96EE13804963F8C2BD6285448198";

//join_result = myLora.initOTAA(appEui, appKey);

```

Locate the OTAA / ABP part in the code. Make changes (uncomment) to use the OTAA join network method, not ABP. Comment out the ABP related code.

13. Upload the code to the Pro Mini and observe the Serial Monitor. If all is configured correct, hardware connected correct and assuming that you have Helium Network coverage, correctly you should see something like this:


```
Startup
When using OTAA, register this DevEUI:
0004A30B00F0FD7F
RN2xx3 firmware version:
RN2483 1.0.5 Oct 31 2018 15:06:52
Trying to join Helium
Successfully joined Helium
TXing
TXing
TXing
TXing
```

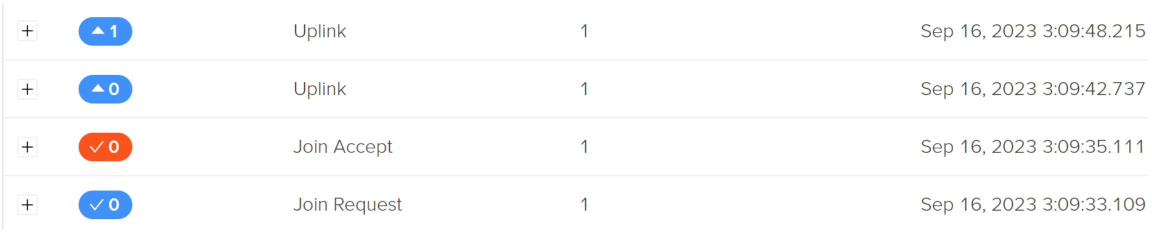
Check your Helium coverage here: https://explorer.helium.com/

You may consider replacing *TTN* with *Helium* in the code, like done above.

Q2: Upload the code and post a screenshot of your serial monitor showing that the device has joined the Helium network. Explain briefly what the code does and what "TXing" means in this context.

Note: It can take up to (20-30 mins) for a sensor (device) to be added and show up on the backend for the first time. I.e. initial transmitted data may be delayed.

14. Assuming that you have a result like above, i.e successfully joined the network, you should be able to see this under Device in the Helium console. Enter the Device menu and click on the device you created. Below the live data you can see an Event log with a Join Request and a Join Accept:

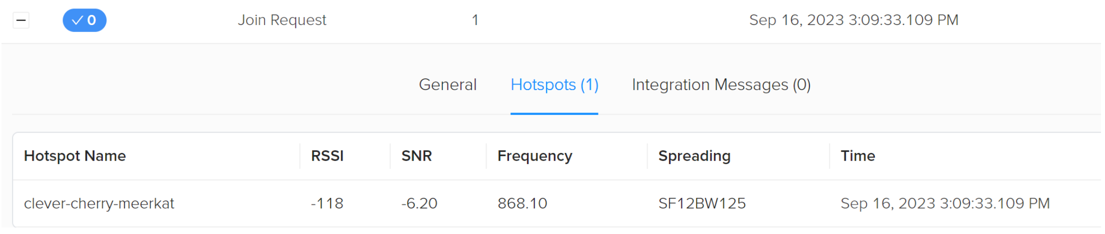


+	▲1	Uplink	1	Sep 16, 2023 3:09:48.215
+	▲0	Uplink	1	Sep 16, 2023 3:09:42.737
+	✓0	Join Accept	1	Sep 16, 2023 3:09:35.111
+	✓0	Join Request	1	Sep 16, 2023 3:09:33.109

You may also see one or more Uplink logs. This basically proves that the LoraWAN message sent from the RN2483-ProMini device has been received at the Helium LoraWAN server.

Q3: Post a screenshot of your Helium backend, showing the Live data, a Join Request and a Join Accept, and perhaps an Uplink event. Note that the first messages sent to the backend may be delayed. Once the device is properly registered in the backend all data will be live data. If your device data is not showing up in the Helium Console skip this part for now and return later.

15. In the Event Log, open a *Join Requests* and check at what spreading factor (SF) the uplink message was sent.

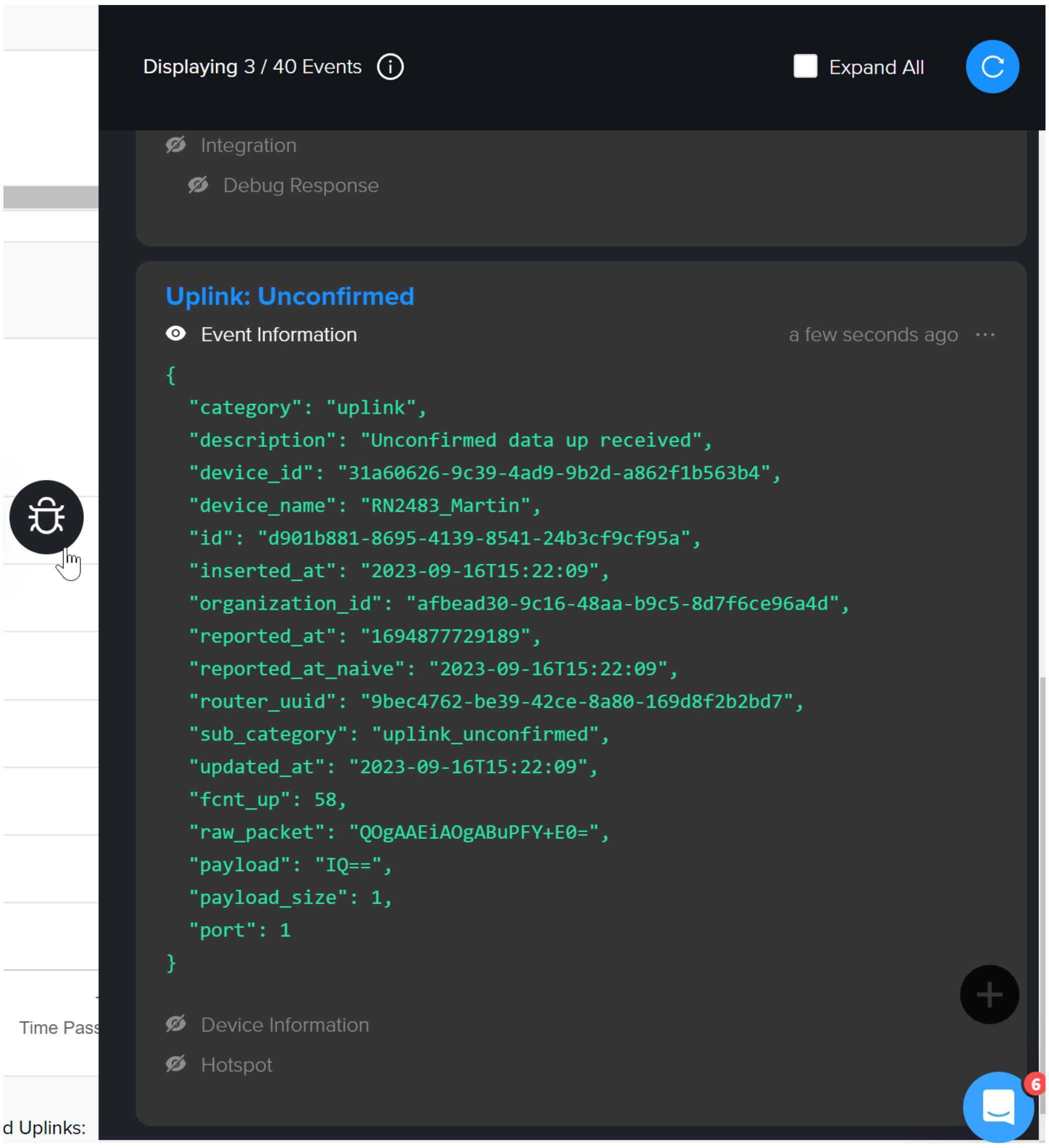


− ✓0 Join Request 1 Sep 16, 2023 3:09:33.109 PM

Hotspot Name	RSSI	SNR	Frequency	Spreading	Time
clever-cherry-meerkat	-118	-6.20	868.10	SF12BW125	Sep 16, 2023 3:09:33.109 PM

Q4: a) At what spreading factor was the Join Request sent? Comment on your findings. Why do you think the spreading factor is low (if low) / high (if high)? Is it a random value or might there be a good reason why it is as it is?
b) Is your message captured by one or more gateways (Helium calls them Hotspots)?
More on spreading factors (SF):
https://www.thethingsnetwork.org/docs/lorawan/spreading-factors/

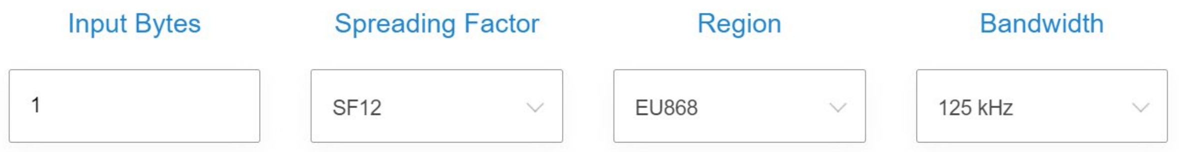
16. When in the device menu, one can click on the *debug-symbol* and see more information about the transmitted message. Below is an example of an Uplink message:



```
{
  "category": "uplink",
  "description": "Unconfirmed data up received",
  "device_id": "31a60626-9c39-4ad9-9b2d-a862f1b563b4",
  "device_name": "RN2483_Martin",
  "id": "d901b881-8695-4139-8541-24b3cf9cf95a",
  "inserted_at": "2023-09-16T15:22:09",
  "organization_id": "afbead30-9c16-48aa-b9c5-8d7f6ce96a4d",
  "reported_at": "1694877729189",
  "reported_at_naive": "2023-09-16T15:22:09",
  "router_uuid": "9bec4762-be39-42ce-8a80-169d8f2b2bd7",
  "sub_category": "uplink_unconfirmed",
  "updated_at": "2023-09-16T15:22:09",
  "fcnt_up": 58,
  "raw_packet": "QOgAAEiAOgABuPFY+E0=",
  "payload": "IQ==",
  "payload_size": 1,
  "port": 1
}
```

Notice the various LoraWAN relevant information that can be seen here (so-called *meta data*). The Payload and Payload size can also be seen here. We may expect the payload to be a "!" or something else that we chose to send in the Arduino code, but it is clearly not that. The Payload is encoded and therefore needs to be decoded and perhaps parsed. More on this in the next section.

Q5: a) Assuming that we are sending 1 byte, use the airtime calculator found here https://www.thethingsnetwork.org/airtime-calculator to calculate the approximate airtime (transmission time) needed to send 1 byte at spreading factor (SF) 12.



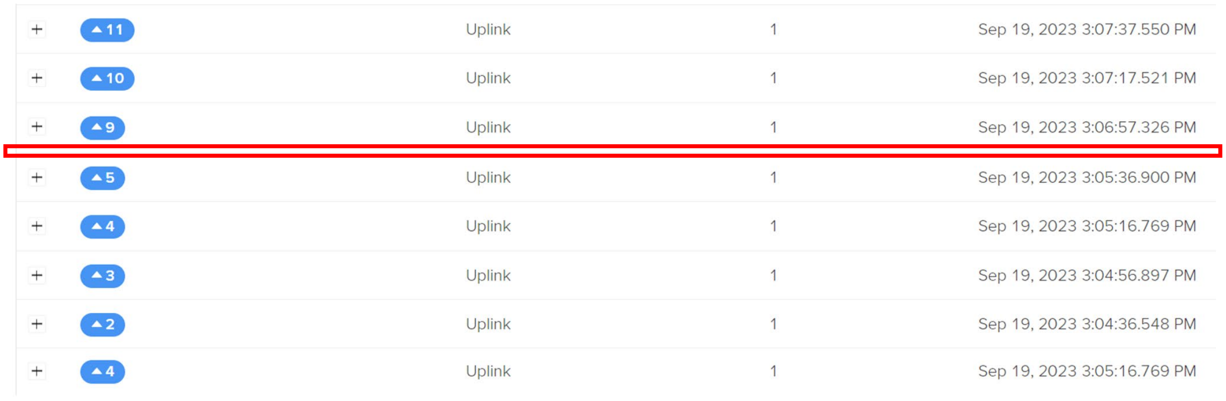
Input Bytes	Spreading Factor	Region	Bandwidth
1	SF12	EU868	125 kHz

b) Considering a duty cycle restriction of 1%, how long does it take before you can send another 1-byte message?
c) How long time does it take to send 24-byte package at SF12 and SF7?

d) What is the theoretically largest payload size that can be sent with LoraWAN?
e) Comment on the above results and discuss the pro's and con's about high vs. low SF.

Q6: a) Send 10 LoraWAN messages with a random payload (e.g. TX1), each message with a separation of 10 seconds. Looking into the Event Log in the Helium console, how many of the messages was successfully received compared to how many TXing attempts was made according to the Serial Monitor?
b) Now repeat the experiment with 30 second intervals. How many did you receive of the 10 messages? Try to comment on the difference – if you see one.
Hint: Spreading factor and duty-cycle restrictions may play a role here.

17. Lost packages. Notice that in situations with poor LoraWAN coverage, a bad antenna or for whatever reason the transmitted message from the device does not reach the Helium backend, you will find missing frames in the Event Log (6-8 are missing below). This information is available because a frame counter *fcnt_up,* is sent with each LoraWAN Uplink transmission.
The *fcnt_up* count can be found in the JSON formatted *meta data* info available in the Debug panel on Helium (example on page 8).



+	▲11	Uplink	1	Sep 19, 2023 3:07:37.550 PM
+	▲10	Uplink	1	Sep 19, 2023 3:07:17.521 PM
+	▲9	Uplink	1	Sep 19, 2023 3:06:57.326 PM
+	▲5	Uplink	1	Sep 19, 2023 3:05:36.900 PM
+	▲4	Uplink	1	Sep 19, 2023 3:05:16.769 PM
+	▲3	Uplink	1	Sep 19, 2023 3:04:56.897 PM
+	▲2	Uplink	1	Sep 19, 2023 3:04:36.548 PM
+	▲4	Uplink	1	Sep 19, 2023 3:05:16.769 PM

Decoding the payload on the Helium Console

At this point, the transmitted Payload goes nowhere and is simply terminated, or lost, at the Helium server. Next, we will set up a decoding and parsing function before finally forwarding the data to a 3rd party service or server. This is needed to retrieve and store data sent from the LoraWAN device.

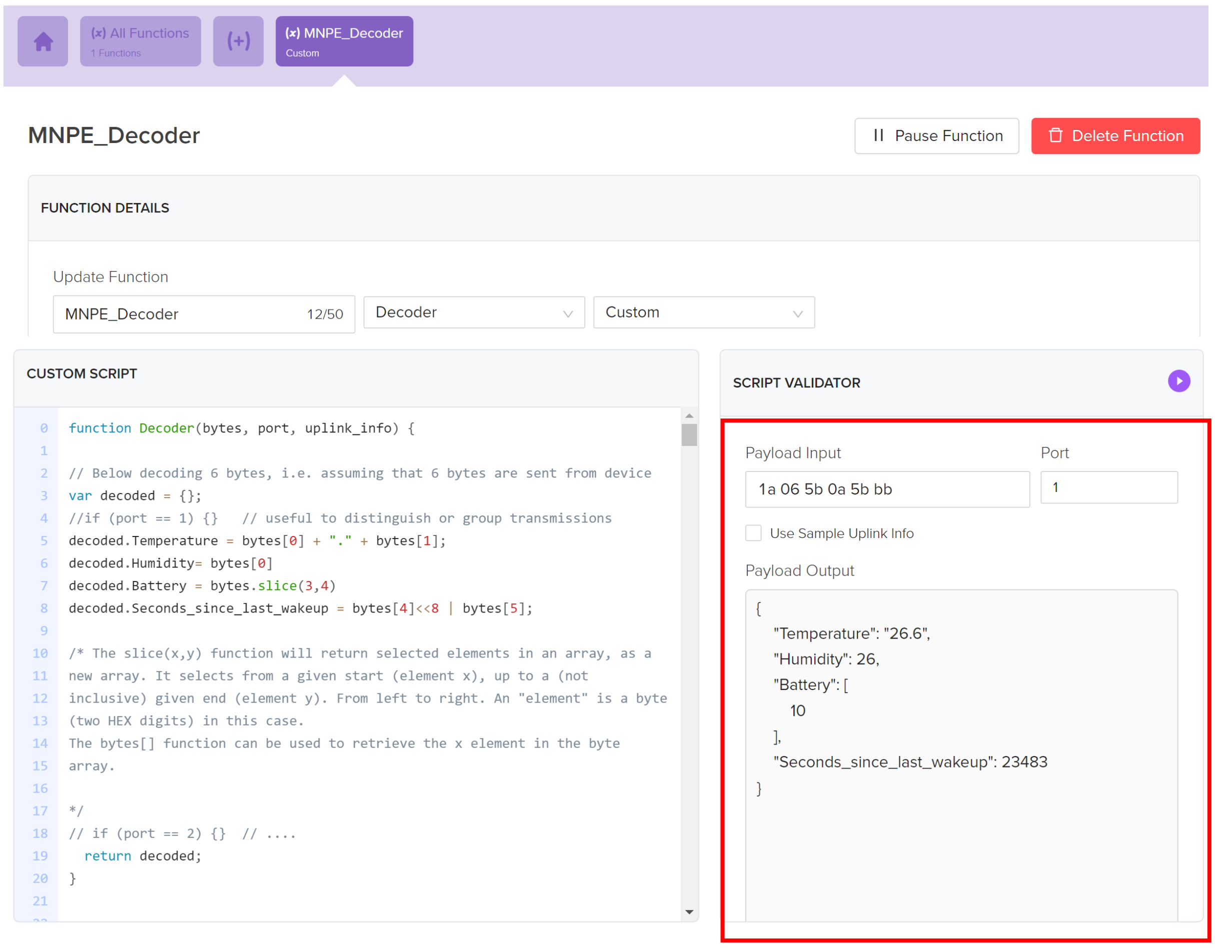
The difference between **decoding** and **parsing** is that *decoding* is to convert from an encrypted form to plain text while *parsing* is to resolve into its elements, for example a JSON object. More on JSON objects here: https://www.w3schools.com/whatis/whatis_json.asp

18. It could be seen above how the payload was in a non-readable format. In fact it is encoded in *base64*[3] so copying the "payload" content into a base64 decoder (https://www.base64decode.org/) you will get "!", or whatever you sent. Base64, is an encoding where 64 characters are used for encoding, compared to HEX which uses 16 characters and is also called base-16.

Q7: Head back to your Arduino code and make changes to transmit your student ID instead of "!". Retrieve the base64 encoded message from the Debug monitor on the Helium Console and use https://www.base64decode.org/ to decode the payload back into plain text. Put a screenshot of both the Debug console showing the payload and the converted payload into the report.

19. Rather than using base64 decoders manually, this can be done somewhat smarter and more automated. Helium gives the possibility to write a custom JavaScript that decodes the payload. Click on Functions in the Helium Console and select to Add a new function. The Decoder script takes as argument a byte array, consisting of the payload from the message we sent. Each element is a byte and is not base64 encoded. I.e. if you from RN2483 send the following HEX data: *1a 06 5b 0a 5b bb,* this would consist of 6 bytes, i.e. six elements in the *bytes* array. An example is given below of how this could be decoded and parsed into a JSON formatted object that can be seen on the lower right in Payload Output (figure below).

[3] https://builtin.com/software-engineering-perspectives/base64-encoding and https://stackoverflow.com/questions/68891169/why-use-base64-in-iot-use-cases



MNPE_Decoder

FUNCTION DETAILS

Update Function: MNPE_Decoder 12/50 | Decoder | Custom

CUSTOM SCRIPT

```
function Decoder(bytes, port, uplink_info) {

// Below decoding 6 bytes, i.e. assuming that 6 bytes are sent from device
var decoded = {};
//if (port == 1) {}   // useful to distinguish or group transmissions
decoded.Temperature = bytes[0] + "." + bytes[1];
decoded.Humidity= bytes[0]
decoded.Battery = bytes.slice(3,4)
decoded.Seconds_since_last_wakeup = bytes[4]<<8 | bytes[5];

/* The slice(x,y) function will return selected elements in an array, as a
new array. It selects from a given start (element x), up to a (not
inclusive) given end (element y). From left to right. An "element" is a byte
(two HEX digits) in this case.
The bytes[] function can be used to retrieve the x element in the byte
array.

*/
// if (port == 2) {}  // ....
  return decoded;
}
```

SCRIPT VALIDATOR

Payload Input: 1a 06 5b 0a 5b bb
Port: 1
Use Sample Uplink Info

Payload Output

```
{
   "Temperature": "26.6",
   "Humidity": 26,
   "Battery": [
      10
   ],
   "Seconds_since_last_wakeup": 23483
}
```

In this example above it would correspond to a transmission of these 6 bytes “1a 06 5b 0a 5b bb” or {0x1a, 0x06, 0x5b, 0x0a, 0x5b, 0xbb} if formatted as a HEX array in Arduino. Again, notice the HEX formatting *0x*, which signifies to the compiler that our values are HEX-formatted. I.e., in decimal it would be (26, 6, 91, 10, 91, 187).

In the example code of the sketch we’re using, the function *myLora.tx()* is used to transmit the LoraWAN message. Earlier, the “!” character was transmitted. From the rn2xx3 Library it can be understood that the *myLora.tx()* function will convert the char “!” to HEX (0x21) and transmit it. I.e. the *myLora.tx()* function expects a *Char type*. More on *data types* here: https://www.tutorialspoint.com/arduino/arduino_data_types.htm
The rn2xx3 library function ***myLora.txBytes()*** can be used to send decimal or bytes.

Example function code (Javascript):

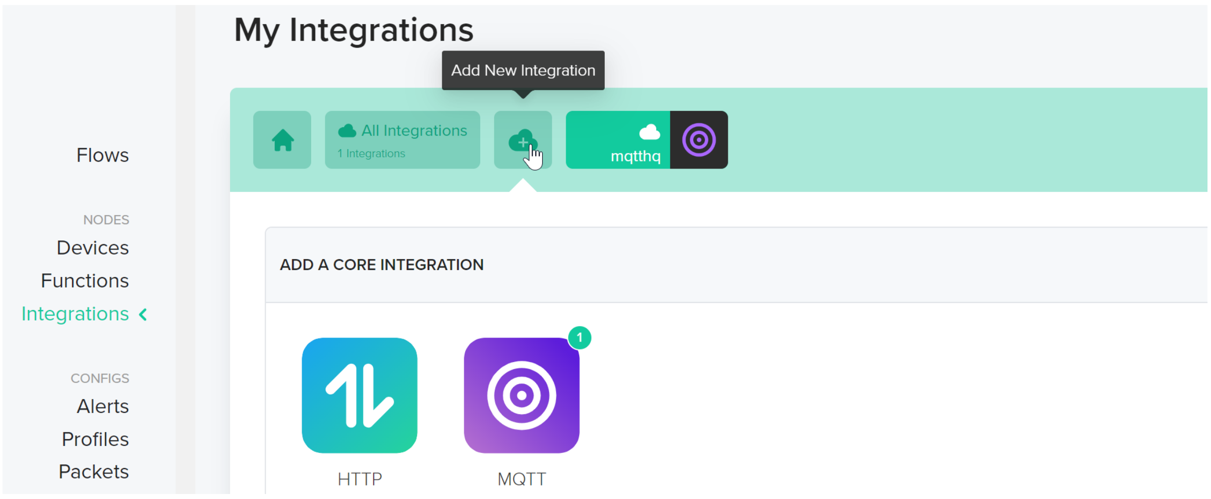
```
function Decoder(bytes, port, uplink_info) {
var decoded = {};
//if (port == 1) {
decoded.Temperature = bytes[0] + "." + bytes[1];
decoded.Humidity= bytes[2]
decoded.Battery = bytes.slice(3,4)
decoded.Seconds_since_last_wakeup = bytes[4]<<8 | bytes[5];
// }
  return decoded;
}
```

Q8: Inspired by the code above, write a simple decoder script in your Helium console (under Function) that can decode a single byte sent to the backend. Let us assume that we're sending a temperature reading (0-255 degrees). Give it a name and save the Function. Test the code with a HEX formatted Payload Input corresponding to a temperature reading between 0 and 100 degrees and post a screenshot (similar to the one above) here.
Hint: Use one of the JavaScript functions described in the screenshot above to retrieve the 0'th (and only) element in the array. Convert the temperature (decimal) to HEX using https://www.rapidtables.com/convert/number/decimal-to-hex.html. Insert the converted HEX into Payload Input to test your function.

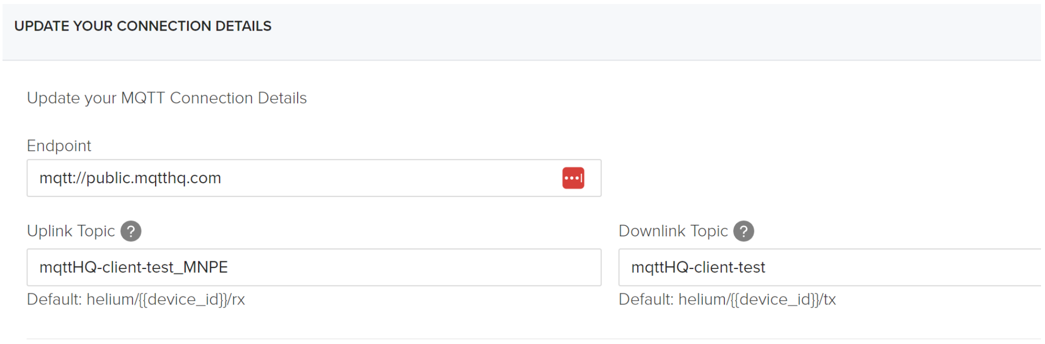
Integrating with an endpoint

As mentioned earlier, the messages sent from the RN2483-ProMini device are currently terminated at the Helium server and not stored. Earlier a decoding and parsing function was made so now it is time to add an endpoint, or Integration as it is called in Helium.

20. Clicking on Integrations in the Helium console, the Integration options that exist can be seen. Click on Add new integration and select MQTT[4].



1. For this quick test / proof-of-concept we will use a MQTT service called MQTTHQ (https://mqtthq.com/), which also offers an online public, insecure, client that can be used for testing: https://mqtthq.com/client. From mqtthq.com you will get the Endpoint details needed to create the Integration on Helium. On https://mqtthq.com/client you can create your own unique topic, to which Helium will post the decoded and passed messages. See below how it could be done.


UPDATE YOUR CONNECTION DETAILS

Update your MQTT Connection Details

Endpoint
mqtt://public.mqtthq.com

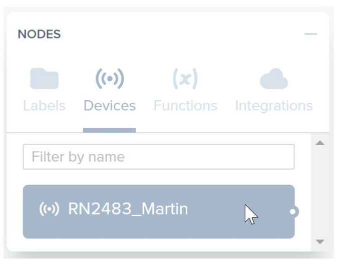
Uplink Topic
mqttHQ-client-test_MNPE
Default: helium/{{device_id}}/rx

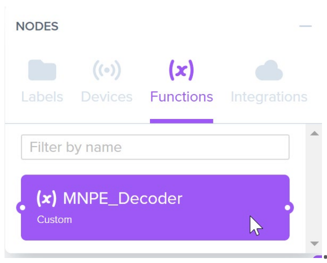
Downlink Topic
mqttHQ-client-test
Default: helium/{{device_id}}/tx

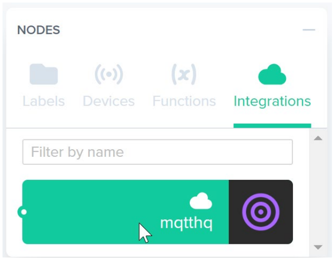
2. With the MQTT Integration created this all needs to be connected. I.e. the data coming from our device, the decoding function and the MQTT integration endpoint. To connect it all, head over to Flows in the Helium Console and click on the + sign in

[4] https://blog.akenza.io/what-is-mqtt & https://www.hivemq.com/mqtt-essentials/

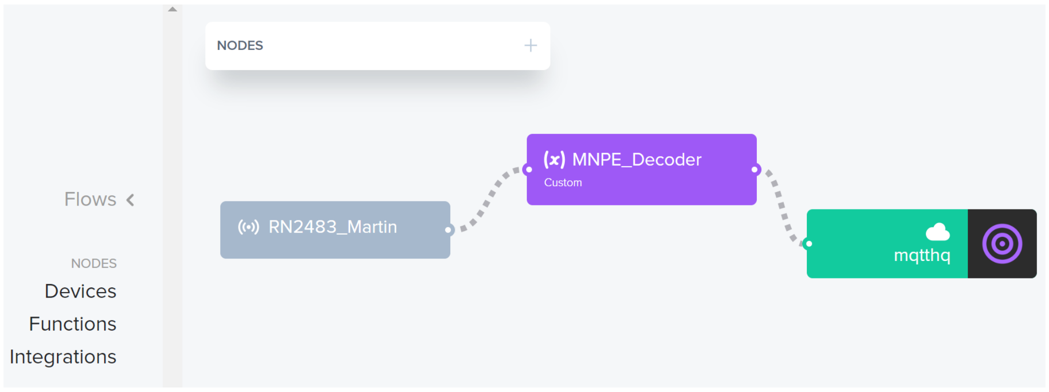
Nodes. Add Device, add the Function created earlier and add the MQTT integration:







Finally, connect them all:



Helium is now ready to receive data sent from your RN2483-ProMini device, decode and parse it, and forward it to the public MQTT server. For a proper use-case one would not use a public client but rather a secure and more professional service, for example HiveMQ (https://www.hivemq.com/).

End-to-end transmission of data over Helium LoraWAN network

Finally, we can do an end-to-end transmission of data over the LoraWAN network. Earlier, the function *myLora.tx("!")* was used to transmit the "!" character. As also mentioned earlier, if we want to transmit decimal numbers, or bytes, we should use *myLora.txBytes()* function instead. From the rn2xx3 library header file we get the following description about the function:

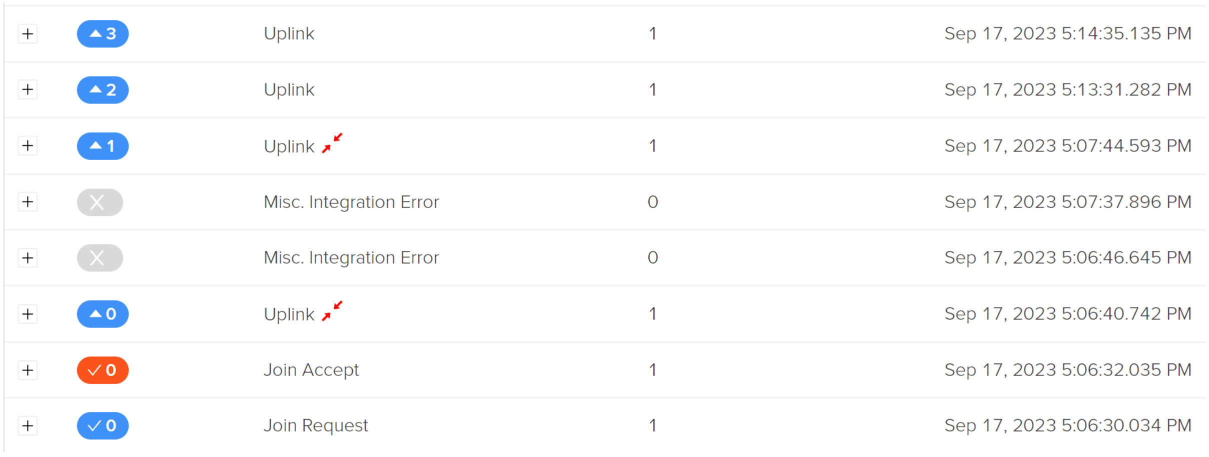
```
/*
 * Transmit raw byte encoded data via LoRa WAN.
 * This method expects a raw byte array as first parameter.
 * The second parameter is the count of the bytes to send.
 */
TX_RETURN_TYPE txBytes(const byte*, uint8_t);
```

Q9: Replace the *myLora.tx()* code with *myLora.txBytes()* in the Arduino code and transmit a hardcoded temperature value of 26 every 60 seconds. There are several ways to do this, the code below can be used for inspiration. Put a screenshot of your code in the report.

```
uint8_t txBuffer[] = {26}
myLora.txBytes(txBuffer,sizeof(txBuffer));
```

If *myLora.tx()* was used to transmit a decimal number, e.g. 26, then 26 would be assumed to be ASCII text, **not** a decimal number. I.e. prior to transmission, it is converted to *0x32 0x36* where *32* and *36* are the HEX representations of the ASCII characters "2" and "6" respectively. When received at the Helium server, *32 and 36* are interpreted as decimal 50 and 52 – quite different from the number 26 we attempted to send.
Long story short, use the function *myLora.txBytes()* to send decimal or bytes.
A table showing the connection between ASCII char, HEX and DEC can be useful for debugging these cases. Here is an example of such a table.

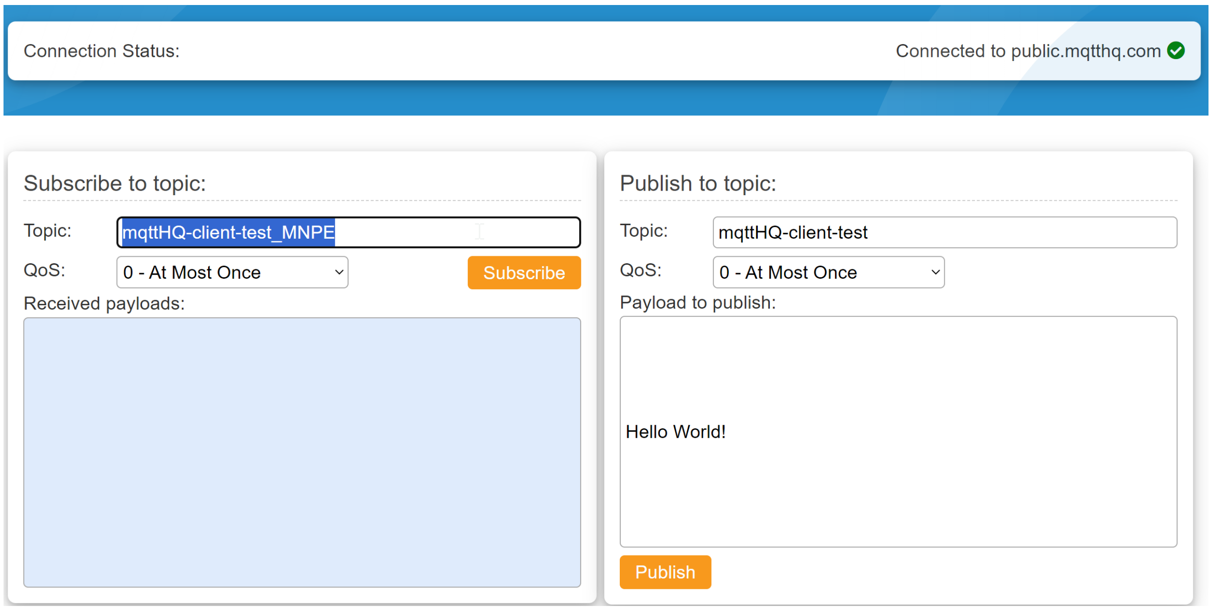
3. After making the above changes and uploading to the Pro Mini, head over to the Helium console and enter the Device menu. If all went well you will, in the Event Log, see a Join Request, Join Accept and eventually, hopefully, also some Uplinks. Notice that there may be "integration errors" and "broken uplinks" in the beginning, like below. This is normal with newly deployed Decode functions and Integrations.



+	▲3	Uplink	1	Sep 17, 2023 5:14:35.135 PM
+	▲2	Uplink	1	Sep 17, 2023 5:13:31.282 PM
+	▲1	Uplink	1	Sep 17, 2023 5:07:44.593 PM
+	✕	Misc. Integration Error	0	Sep 17, 2023 5:07:37.896 PM
+	✕	Misc. Integration Error	0	Sep 17, 2023 5:06:46.645 PM
+	▲0	Uplink	1	Sep 17, 2023 5:06:40.742 PM
+	✓0	Join Accept	1	Sep 17, 2023 5:06:32.035 PM
+	✓0	Join Request	1	Sep 17, 2023 5:06:30.034 PM

Q10: Post a screenshot, like above of your Event Log.

4. Once the integration errors resolve, head over to https://mqtthq.com/client and enter the Topic you chose earlier when creating the MQTT integration, and click Subscribe. Example:

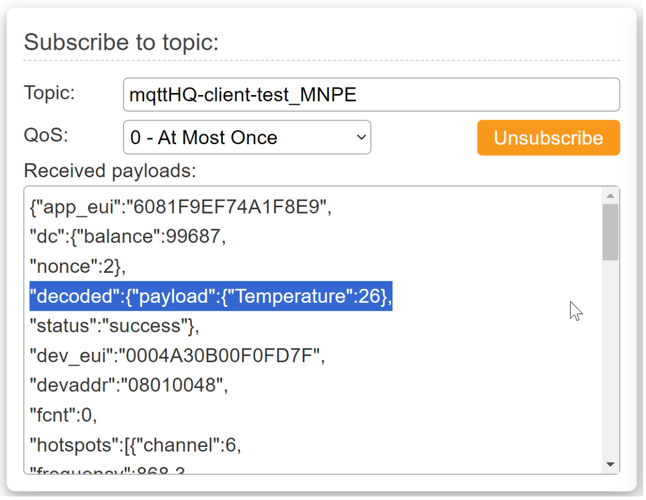

Connection Status: Connected to public.mqtthq.com

Subscribe to topic:
Topic: mqttHQ-client-test_MNPE
QoS: 0 - At Most Once

Received payloads:

Publish to topic:
Topic: mqttHQ-client-test
QoS: 0 - At Most Once
Payload to publish:
Hello World!

5. Now, once a LoraWAN message arrives at the Helium server, this message is decoded, parsed and finally Published to your Topic on that page. Finally, you should see something like this – notice the decoded part of the message:


Subscribe to topic:
Topic: mqttHQ-client-test_MNPE
QoS: 0 - At Most Once

Received payloads:

```
{"app_eui":"6081F9EF74A1F8E9",
"dc":{"balance":99687,
"nonce":2},
"decoded":{"payload":{"Temperature":26},
"status":"success"},
"dev_eui":"0004A30B00F0FD7F",
"devaddr":"08010048",
"fcnt":0,
"hotspots":[{"channel":6,
```

Q11: Post a screenshot, like the above from mqttHQ, showing your published message from your device.

This is the very basics of sending a LoraWAN message from the device to a backend utilizing MQTT. There are a great number of alternative integration options; Azure, AWS, Google Cloud, Datacake etc. All offering a number of additional services such as data storage, databases, data analytics, visualizations, mobile App integration etc. Guides for all of them can be found via Google.

Q12: Make a code-integration with the e-paper and temperature sensor setup from last time. Send the temperature measurements made with the analog sensor to the Helium backend and MQTT broker and verify the measurement are the same on the e-paper screen. Document with screenshots and a photo of the sensor.

Sending and encoding multiple values in a byte array

In the above example, a single temperature measurement was sent, a single byte. Typically we want to send more information than that. Below is described how that can be done.

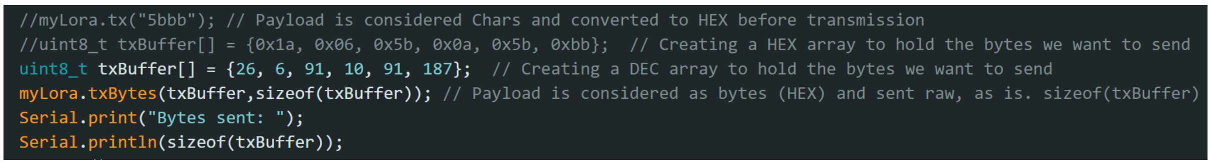
6. As an example, one can imagine having a device capable of measuring Temperature, Humidity, Battery level time passed since last wake-up. Imagine the following has been measured at the device:

   Temperature: 26.6 deg.
   Humidity: 91%
   Battery level (0-10): 10
   Time since last wakeup: 23483 seconds

   To save transmission power/battery, duty-cycle (time-on-air) and in general be efficient, we could break up the data and transmit it as (26, 6, 91, 10, 91, 187) and use the Decoder function and parser code from the previous page to reconstruct the information.

   Note, the number *23483* is constructed by using the last two bytes (0x5b, 0xbb) as a single byte can only have 256 different values. To reconstruct the number, 0x5b is left shifted 8 bits followed by a bit-wise *or* operation[5] with 0xbb to achieve 23483 (0x5bbb).

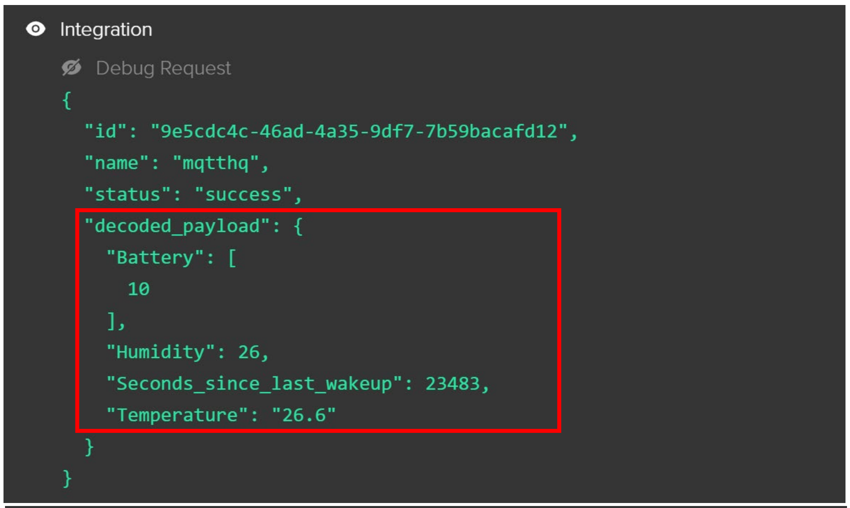
   The Arduino code to transmit could look like this:



```
//myLora.tx("5bbb"); // Payload is considered Chars and converted to HEX before transmission
//uint8_t txBuffer[] = {0x1a, 0x06, 0x5b, 0x0a, 0x5b, 0xbb};  // Creating a HEX array to hold the bytes we want to send
uint8_t txBuffer[] = {26, 6, 91, 10, 91, 187};  // Creating a DEC array to hold the bytes we want to send
myLora.txBytes(txBuffer,sizeof(txBuffer)); // Payload is considered as bytes (HEX) and sent raw, as is. sizeof(txBuffer)
Serial.print("Bytes sent: ");
Serial.println(sizeof(txBuffer));
```

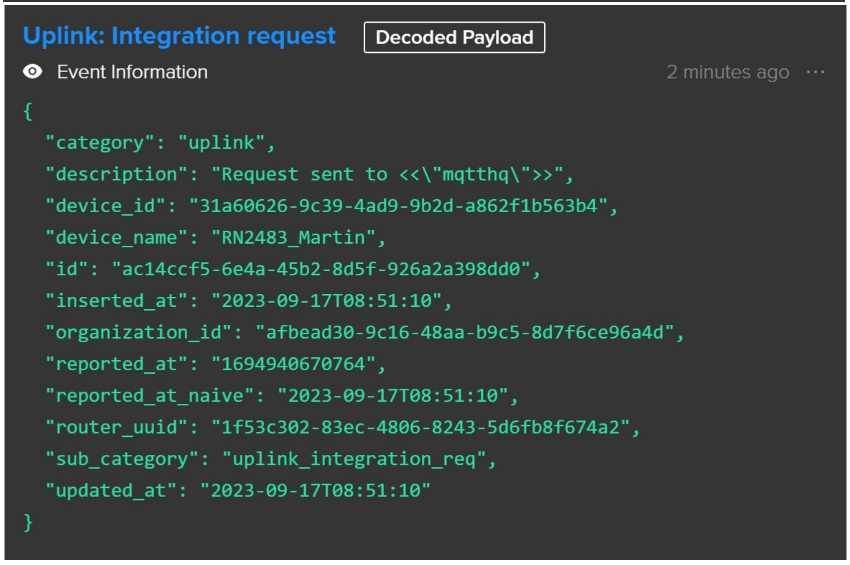
   If myLora.txBytes() is used to send the payload (26, 6, 91, 10, 91, 187), plus using the decode/parser function from above, one will receive something like this in the Debugger:

[5] https://docs.arduino.cc/learn/programming/bit-math


Integration

Debug Request

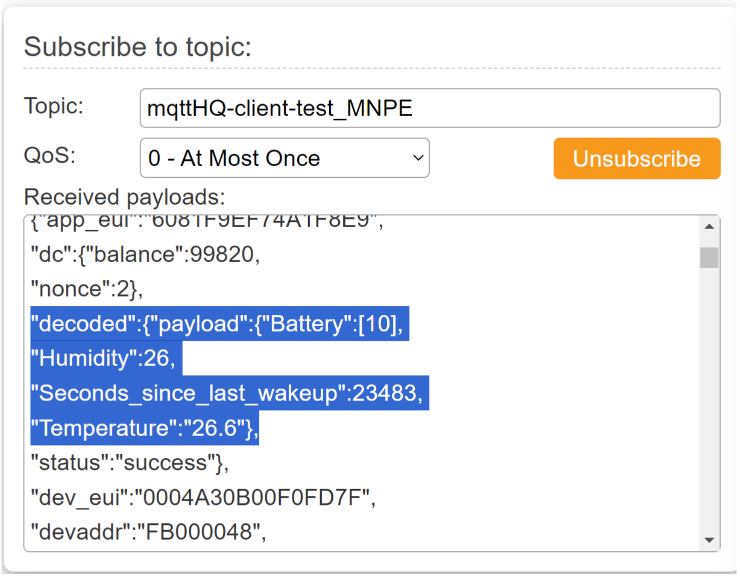
```
{
  "id": "9e5cdc4c-46ad-4a35-9df7-7b59bacafd12",
  "name": "mqtthq",
  "status": "success",
  "decoded_payload": {
    "Battery": [
      10
    ],
    "Humidity": 26,
    "Seconds_since_last_wakeup": 23483,
    "Temperature": "26.6"
  }
}
```


Uplink: Integration request Decoded Payload

Event Information 2 minutes ago

```
{
  "category": "uplink",
  "description": "Request sent to <<\"mqtthq\">>",
  "device_id": "31a60626-9c39-4ad9-9b2d-a862f1b563b4",
  "device_name": "RN2483_Martin",
  "id": "ac14ccf5-6e4a-45b2-8d5f-926a2a398dd0",
  "inserted_at": "2023-09-17T08:51:10",
  "organization_id": "afbead30-9c16-48aa-b9c5-8d7f6ce96a4d",
  "reported_at": "1694940670764",
  "reported_at_naive": "2023-09-17T08:51:10",
  "router_uuid": "1f53c302-83ec-4806-8243-5d6fb8f674a2",
  "sub_category": "uplink_integration_req",
  "updated_at": "2023-09-17T08:51:10"
}
```

Or, looking at the MQTT server it would look like this:


Subscribe to topic:

Topic: mqttHQ-client-test_MNPE

QoS: 0 - At Most Once

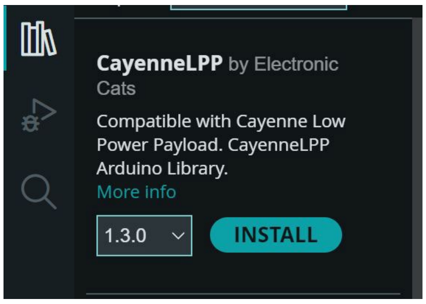
Received payloads:

```
{"app_eui":"6081F9EF74A1F8E9",
"dc":{"balance":99820,
"nonce":2},
"decoded":{"payload":{"Battery":[10],
"Humidity":26,
"Seconds_since_last_wakeup":23483,
"Temperature":"26.6"},
"status":"success"},
"dev_eui":"0004A30B00F0FD7F",
"devaddr":"FB000048",
```

Q13: a) Which values/measurement would you find relevant to send to the backend?
b) Construct an example array for this, like above, *txBuffer = (26, 6, 91, 10, 91, 187) and explain the contents.*

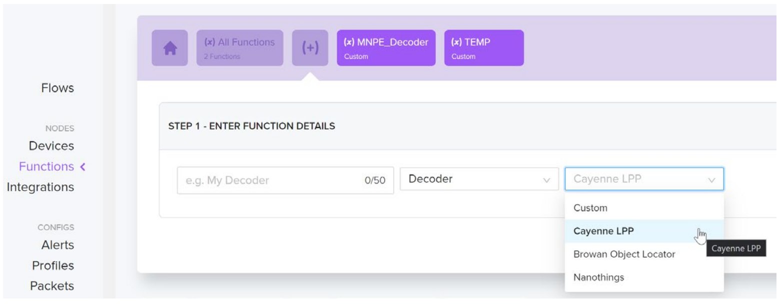
Further compressing data for saving power

Sending data as done above, reduced to bytes, and then manually parsed into a JSON can be automated and optimized by using CayenneLPP (Cayenne Low Power Payload) encoding. A CayenneLPP library can be found and installed in Arduino IDE, and is also supported as a standard decoding format in Helium Functions.



CayenneLPP by Electronic Cats

Compatible with Cayenne Low Power Payload. CayenneLPP Arduino Library.



STEP 1 - ENTER FUNCTION DETAILS

Decoder

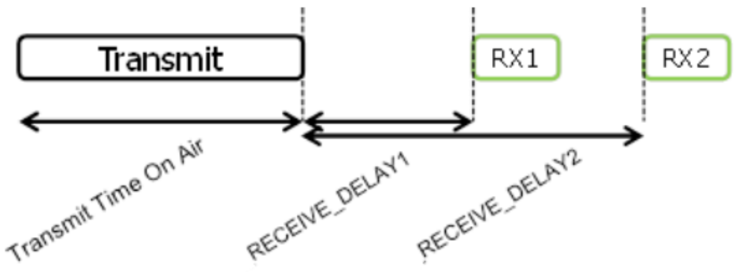
Cayenne LPP

More info on the CayenneLPP library can be found here: https://github.com/ElectronicCats/CayenneLPP and https://www.thethingsnetwork.org/docs/devices/arduino/api/cayennelpp/

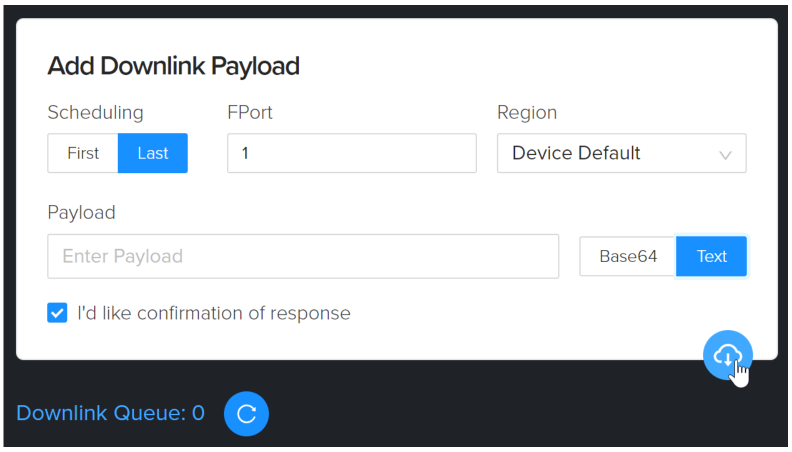
Sending data down-link (server to device) using LoraWAN

Up until now data has only been sent uplink, to Helium or the cloud, if you will. Technically speaking though, some downlink communication was taking place in the *Join Request – Join Accept* procedure but not any data we have control over. In this next part it will be demonstrated how data can be sent to the device, from Helium or via the MQTT broker using Publish/Subscribe.

LoraWAN only allows data to be sent, and received by the device, when the device has sent data itself. I.e. the only opportunity for the device to receive data is after the device transmission ends. After transmission, two receive windows open and this is the only occasion the end-device as to receive data. I.e. messages may be waiting at the Helium server for minutes or hours, before they can be delivered to the device.



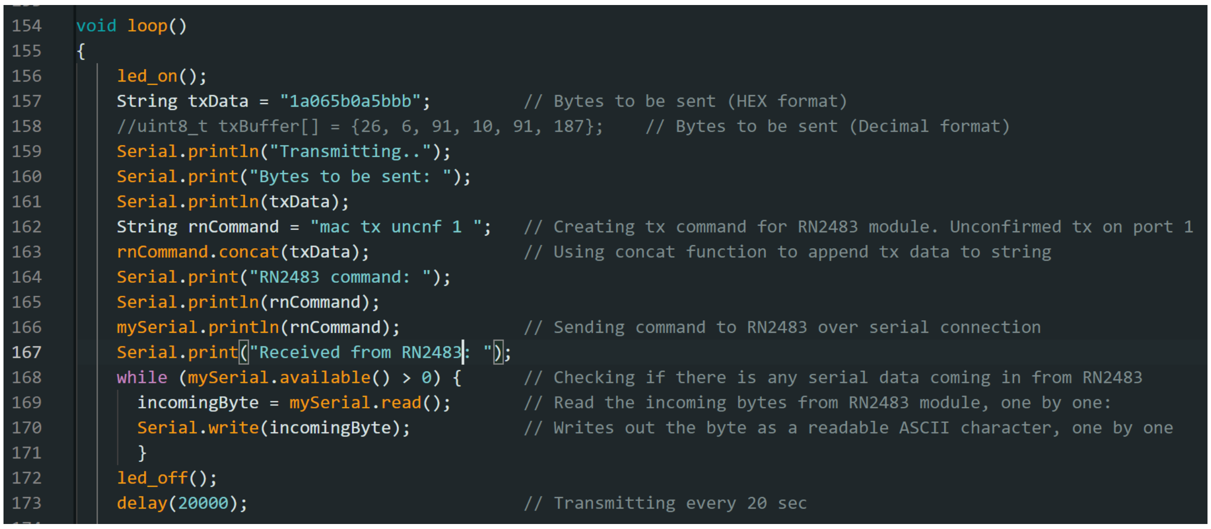
7. In the Helium console, it is rather easy to initiate a downlink session. Enter the Device menu, click on a device and on the right side there is a Downlink icon: Clicking this opens a menu where Text or base64 encoded payload can be sent to the device, after the next uplink.



The challenge now is to extract the received message on the device.

8. The rn2xx3 library does have a function, *myLora.getRX()* function, that will capture the downlink message. This function however will return the incoming message as a String which can cause some type casting trouble. This method is described in Appendix A.
Instead, we can develop our own code rather than relying on the library. Some example code can be seen below. The code uses *raw commands* taken from the RN2483 user manual. A command such as *mac tx unconf 1 4B3A* will transmit an unconfirmed message with the HEX data 4B3A on port 1. Similarly, *mac tx conf* will

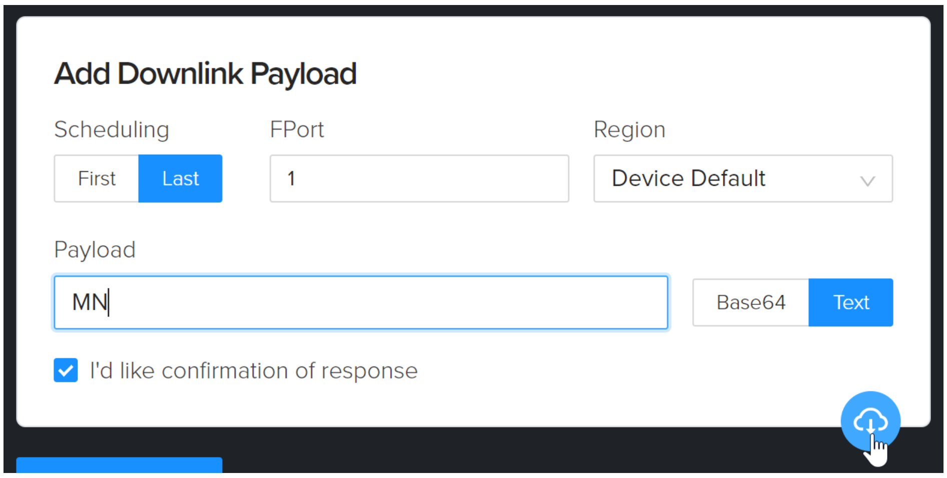
transmit and request a confirmation receipt from the LoraWAN server (Helium). Any responses from RN2483, or incoming data, will be read into the *incomingbyte* variable, and in turn be posted on the serial monitor.



```
void loop()
{
  led_on();
  String txData = "1a065b0a5bbb";          // Bytes to be sent (HEX format)
  //uint8_t txBuffer[] = {26, 6, 91, 10, 91, 187};    // Bytes to be sent (Decimal format)
  Serial.println("Transmitting..");
  Serial.print("Bytes to be sent: ");
  Serial.println(txData);
  String rnCommand = "mac tx uncnf 1 ";   // Creating tx command for RN2483 module. Unconfirmed tx on port 1
  rnCommand.concat(txData);               // Using concat function to append tx data to string
  Serial.print("RN2483 command: ");
  Serial.println(rnCommand);
  mySerial.println(rnCommand);            // Sending command to RN2483 over serial connection
  Serial.print("Received from RN2483: ");
  while (mySerial.available() > 0) {      // Checking if there is any serial data coming in from RN2483
    incomingByte = mySerial.read();       // Read the incoming bytes from RN2483 module, one by one:
    Serial.write(incomingByte);           // Writes out the byte as a readable ASCII character, one by one
    }
  led_off();
  delay(20000);                           // Transmitting every 20 sec
```

Q14: Use this code as inspiration for creating your own code capable of receiving a downlink message from the Helium LoraWAN server. Post a screenshot in the report.

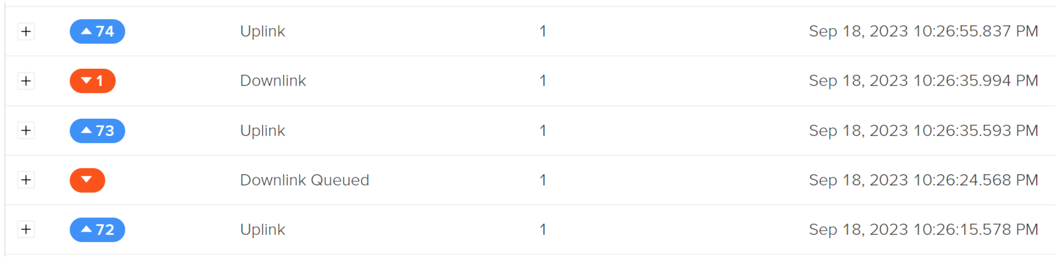
9. Upload the code and verify that the uplink messages are arriving at the Helium console. Use the Downlink feature to send a short message downlink to the ProMini.



Add Downlink Payload

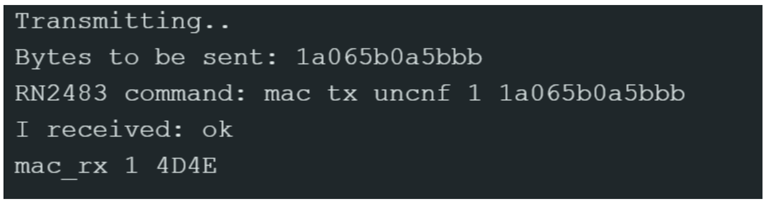
Scheduling: First [✓ Last]
FPort: 1
Region: Device Default
Payload: MN
Base64 [✓ Text]
[✓] I'd like confirmation of response

After a short while you should see your Downlink being queued in the Event Log. A minute or two later you should also see the Downlink message confirming that the server has attempted to deliver the message.



+	▲74	Uplink	1	Sep 18, 2023 10:26:55.837 PM
+	▼1	Downlink	1	Sep 18, 2023 10:26:35.994 PM
+	▲73	Uplink	1	Sep 18, 2023 10:26:35.593 PM
+	▼	Downlink Queued	1	Sep 18, 2023 10:26:24.568 PM
+	▲72	Uplink	1	Sep 18, 2023 10:26:15.578 PM

Finally, in the Serial monitor, the received data can be observed:



```
Transmitting..
Bytes to be sent: 1a065b0a5bbb
RN2483 command: mac tx uncnf 1 1a065b0a5bbb
I received: ok
mac_rx 1 4D4E
```

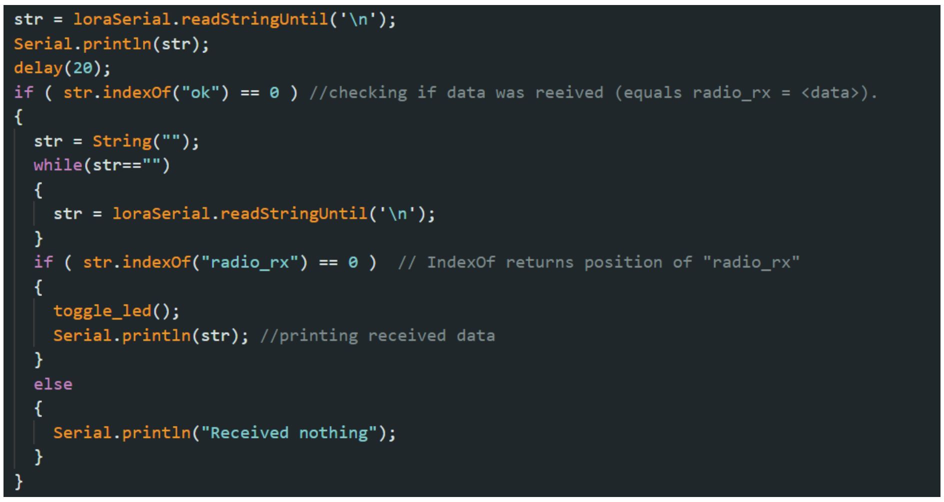
It may take a few tries (Uplink transmissions) for the Downlink message to arrive. Notice how *MN* was sent from Helium console but *4D 4E* is received.

Q15: a) Post screenshots of your own Downlink transmission. From the Event Log as well as the Serial Monitor.
b) Explain why the characters *MN* ended up as *4D 4E*.

For use-case reasons, IoT devices are typically made to send Uplink messages most of the time. This would often be to report various sensor measurements. Downlink messages (to the device) are more rare but can be useful to change the state of a device. It could be to change the Uplink frequency, how often something is measured, or it could be to initiate a GPS measurement, to activate a buzzer or something else. Although challenging, even the device (MCU) firmware can be updated through a number of Downlink messages.

Q16 (OPTIONAL): a) Prepare the Arduino code to receive and handle a Downlink command. Make the code such that sending a certain character or number to the ProMini will change the delay in the transmission loop. For example, sending a "1" will set the transmission delay to 10 seconds, "2" will set it to 1 minute and "3" will set it to 5 minutes.
Send commands (1, 2 or 3) from the Helium Downlink console to test the functionality. Keep in mind that "1,2 and 3" may not end up as "1, 2 and 3" on your device but perhaps this does not have to be a problem..

Code inspiration related to extracting a received message below:

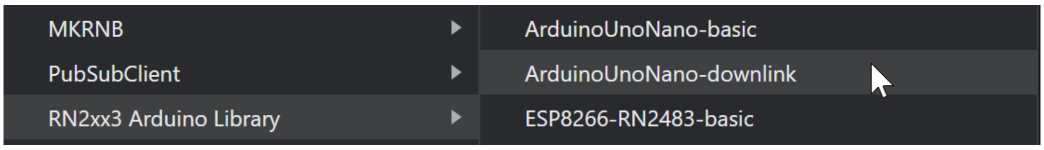


```
str = loraSerial.readStringUntil('\n');
Serial.println(str);
delay(20);
if ( str.indexOf("ok") == 0 ) //checking if data was reeived (equals radio_rx = <data>).
{
  str = String("");
  while(str=="")
  {
    str = loraSerial.readStringUntil('\n');
  }
  if ( str.indexOf("radio_rx") == 0 )  // IndexOf returns position of "radio_rx"
  {
    toggle_led();
    Serial.println(str); //printing received data
  }
  else
  {
    Serial.println("Received nothing");
  }
}
```

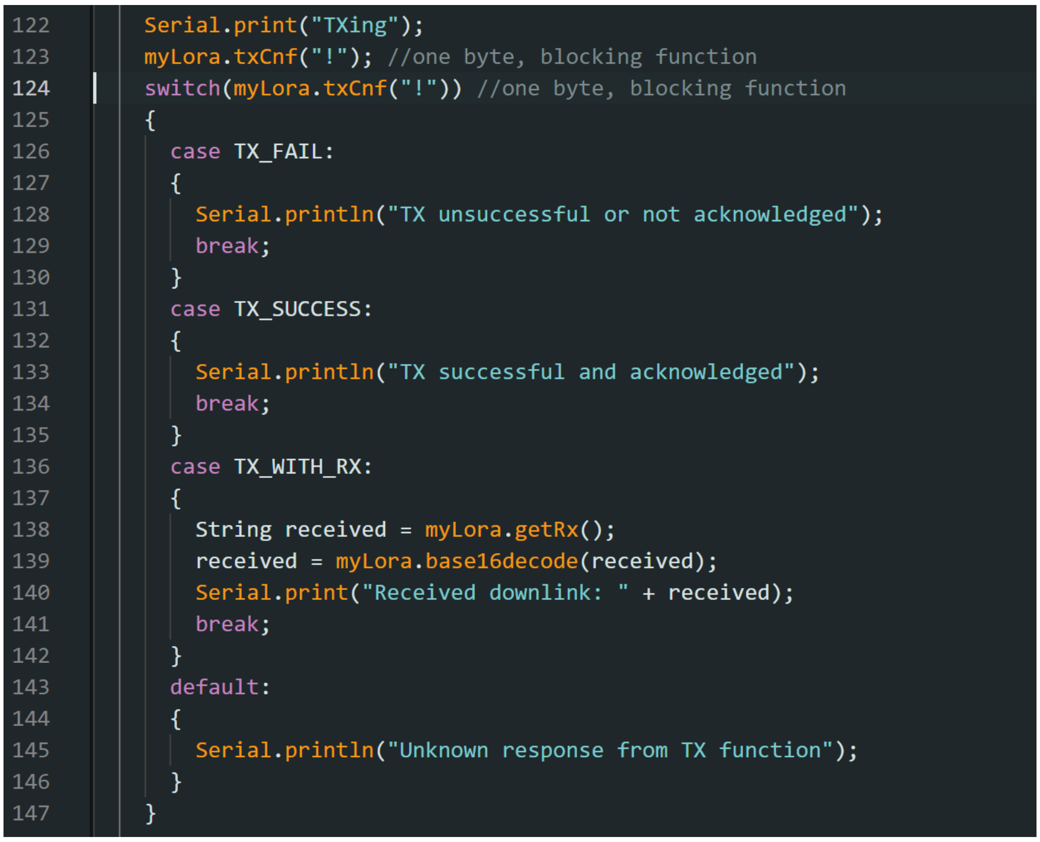
Remember:
This exercise will be bundled with next week's exercise. I.e. deadline is at Oct 4th.

Appendix A: Using the rn2xx3 library receive function

10. If you prefer to use the library example, open the Sketch called ArduinoUnoNano-downlink:



In that file there is “TXing session” that corresponds a lot to the previous sketch example that was used. A function called myLora.txCnf() is used to transmit “!”, followed by a switch statement.



```
122    Serial.print("TXing");
123    myLora.txCnf("!"); //one byte, blocking function
124    switch(myLora.txCnf("!")) //one byte, blocking function
125    {
126      case TX_FAIL:
127      {
128        Serial.println("TX unsuccessful or not acknowledged");
129        break;
130      }
131      case TX_SUCCESS:
132      {
133        Serial.println("TX successful and acknowledged");
134        break;
135      }
136      case TX_WITH_RX:
137      {
138        String received = myLora.getRx();
139        received = myLora.base16decode(received);
140        Serial.print("Received downlink: " + received);
141        break;
142      }
143      default:
144      {
145        Serial.println("Unknown response from TX function");
146      }
147    }
```

Copy that switch statement into your other Arduino sketch and replace the function to the byte-transfer function used above, myLora.txBytes(). Save the file as a new sketch. The code in line 139 above will also have to be commented out for it to function. The base16 decoding function does not work on the data type coming from Helium.

In the above code, the *myLora.getRX()* function will capture any downlink message sent from the LoraWAN server and load it into a String variable called *received.*

Appendix B: About duty cycle and RN2483, and how to by-pass it

I'm getting a `no_free_ch` response from the module. How can I eliminate it?

This is entirely a duty cycle issue, which is regulatory. Generally, the duty cycle limit is 1% (but it does vary sometimes by frequency band and region) and RN2483 keeps "on air" and "off air" timers for each channel to help ensure that you don't violate the regulations. By default, there are three channels enabled, each with 0.33% duty cycle allocation. When pseudo-randomly hopping around the channels, the algorithm looks at the next channel and decides if the timers have enough remaining duty cycle allocation for the transmission. If not, then it skips that channel and moves to the next. If you've used all the allocation for all channels, you get the no_free_ch response. This is quite a common problem during development/learning/debugging, where you tend to send messages far more frequently than normal. In real "sleepy" applications it is rarely an issue. The no_free_ch means that all channels are busy transmitting and you have to wait until you can transmit again.

You can cheat the safeguards just by raising the duty cycle limit for each channel using the command mac set ch dcycle (from the **"RN2483 LoRa® Technology Module Command Reference User's Guide"**, which could be found on the RN2483 product page under the "Documents" section) (https://www.microchip.com/wwwproducts/en/RN2483)

The value that needs to be configured can be obtained from the actual duty cycle X (in percentage) using the following formula: = (100/X) - 1

- mac set ch dcycle 0 9 sets channel 0 to 10% (= 90% off)
- mac set ch dcycle 1 99 sets channel 1 to 1% (= 99% off)
- mac set ch dcycle 2 999 sets channel 2 to 0.1% (= 99.9% off).

Note: This is only allowed for prototyping and research purposes. Any product with modified duty cycles would not pass a CE approval.