Conception d'un protocole adapté aux objets connectés dans un réseau dense

Timothée [redacted]
Candidat MPI/1[redacted]8

TIPE
Informatique Pratique
Informatique Théorique

2023

Introduction

1. Pourquoi ce projet ?
2. Objectifs :
    1. Proposer un protocole et son implémentation pour répondre à ce cas particulier.
    2. Conclure par une phase expérimentale de son intêret.
3. Découvrons les étapes de réalisation de ce projet
    1. Cahier des charges
    2. Techniques proposées, étudiées, et implémentées
    3. Conception puis implémentation du protocole
    4. Mesures et expérimentations

Cahier des charges du protocole

1. Adapté aux objets connectés
   1. Conscient du coût des transmissions réseaux
   2. Implémentable sur système embarqué
2. Adapté à la ville (réseau dense)
   1. Respectueux des réglementations
   2. Permettant des transmissions efficaces même dans un environnement dense
3. Quelques attentes personnelles
   1. Permettant la retransmission en cas d'erreur
   2. Accomplissant les échanges au niveau du lien (et pas plus)

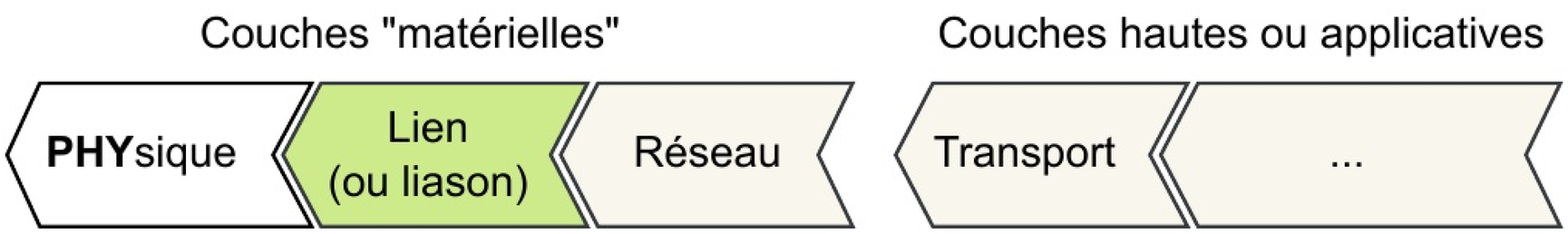


Figure – Modèle OSI de la pile réseau

Considérations techniques

Choix propres à notre projet :

1. Réseau et protocole LoRa
    1. Adaptée aux transmissions à faible débit
    2. Une base simple et permissive
2. Radio RFM95W
3. Carte de développement ESP32

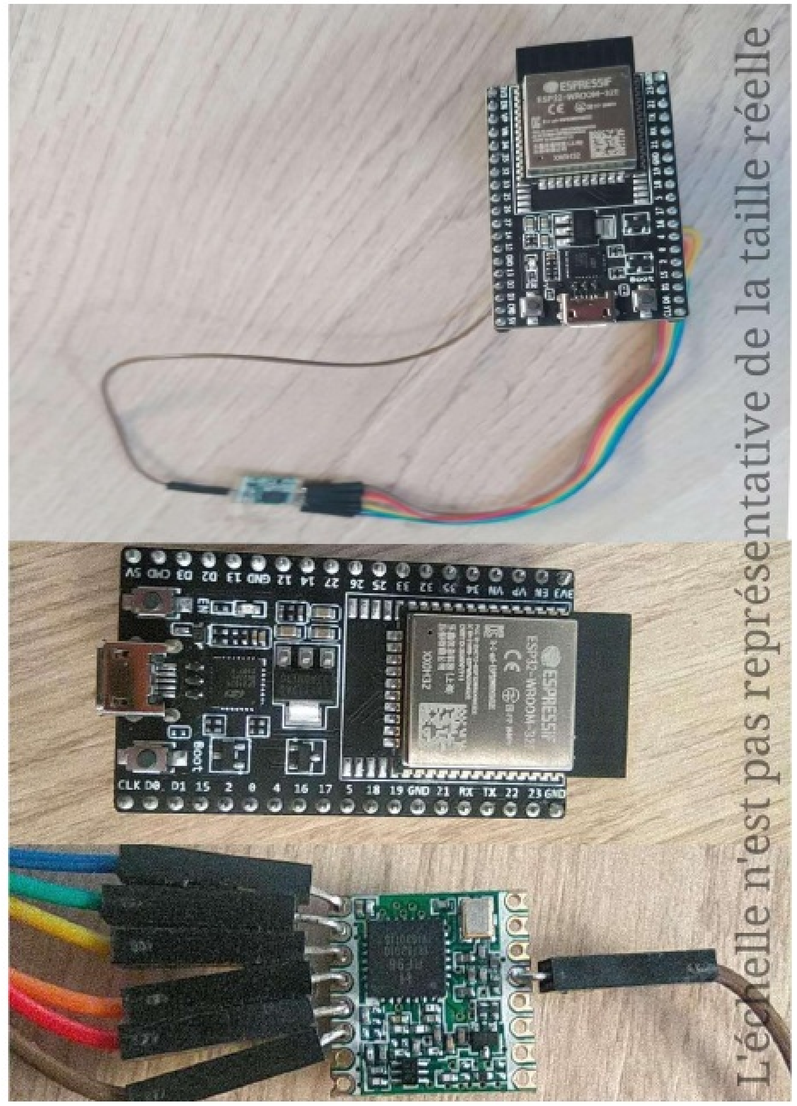


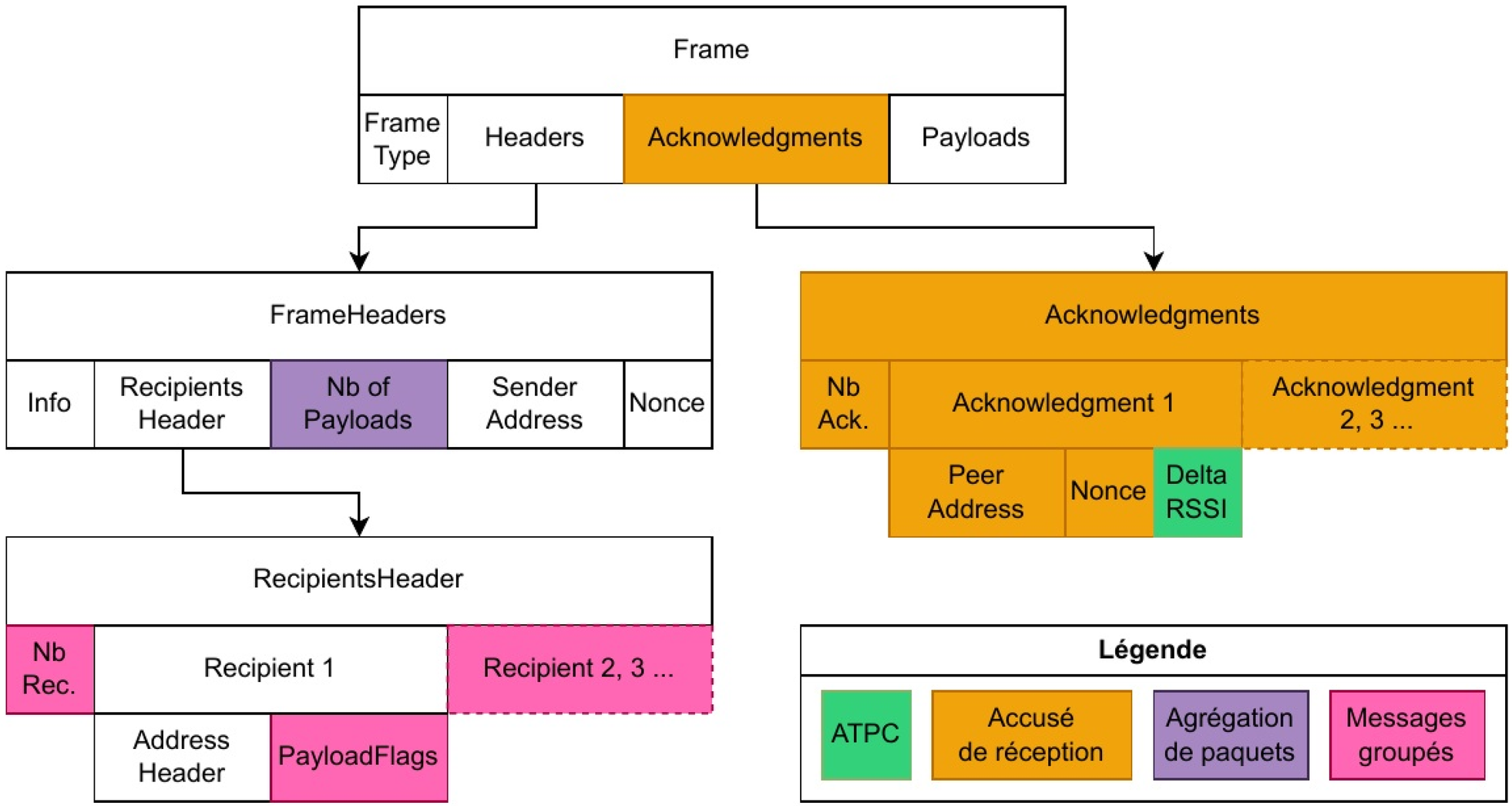
Figure – Un module ESP32 et son module radio RFM95W

Étude des techniques

1. Agrégation des paquets
2. Messages groupés
3. Accusé de réception par bloc (*Block Acknowledgment*)
4. Canaux (fréquences) rotatifs
5. *Adaptive Transmission Power Control* (ATPC)
   1. Méthode active (nécessitant une boucle de rétroaction)
   2. Permet une adaptation fine à l'environnement

Point conception

- Création de trames propres au protocole
- Conception de l'ATPC réutilisant les structures déjà implémentées.



Point implémentation

- Contrainte : notre protocole doit hypothétiquement être utilisable
- Séparation explicite des actions sur la radio
- Interface *callback* optionnelle
- Écriture de patchs pour la bibliothèque `rust-radio-sx127x`

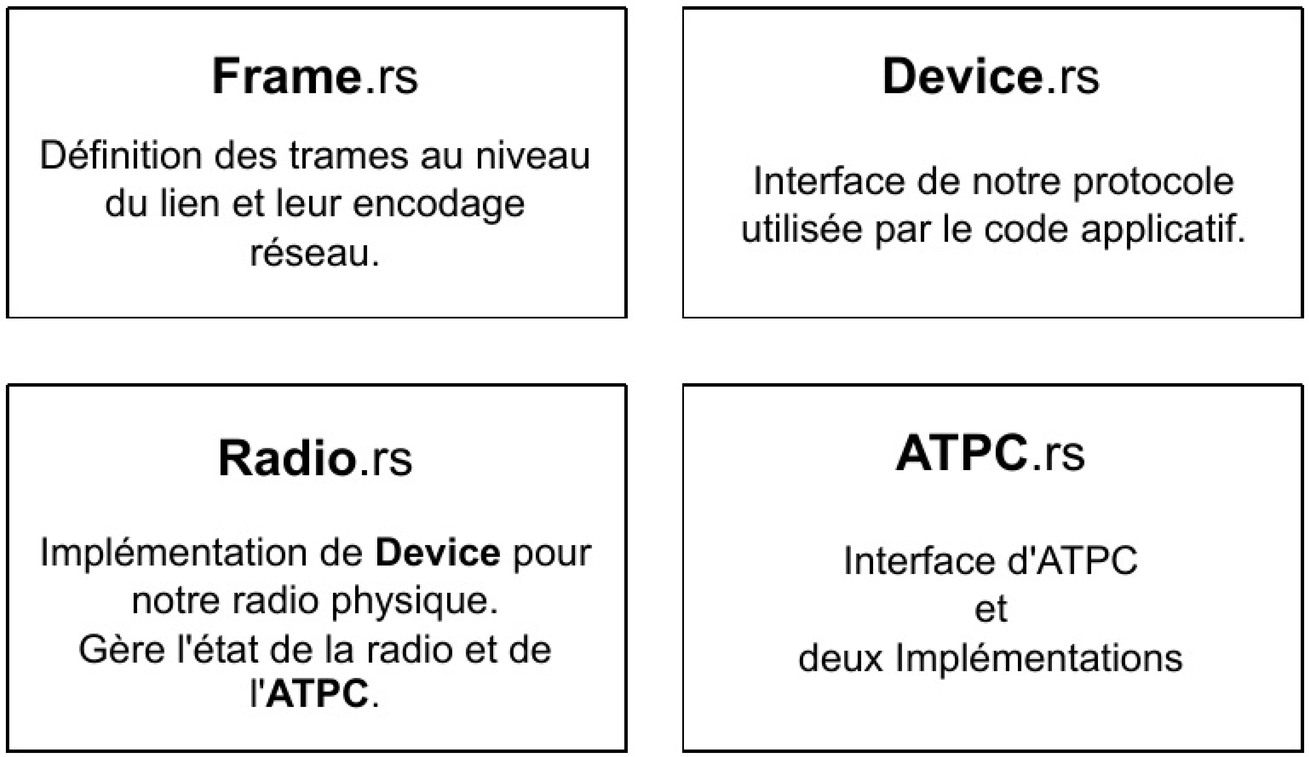
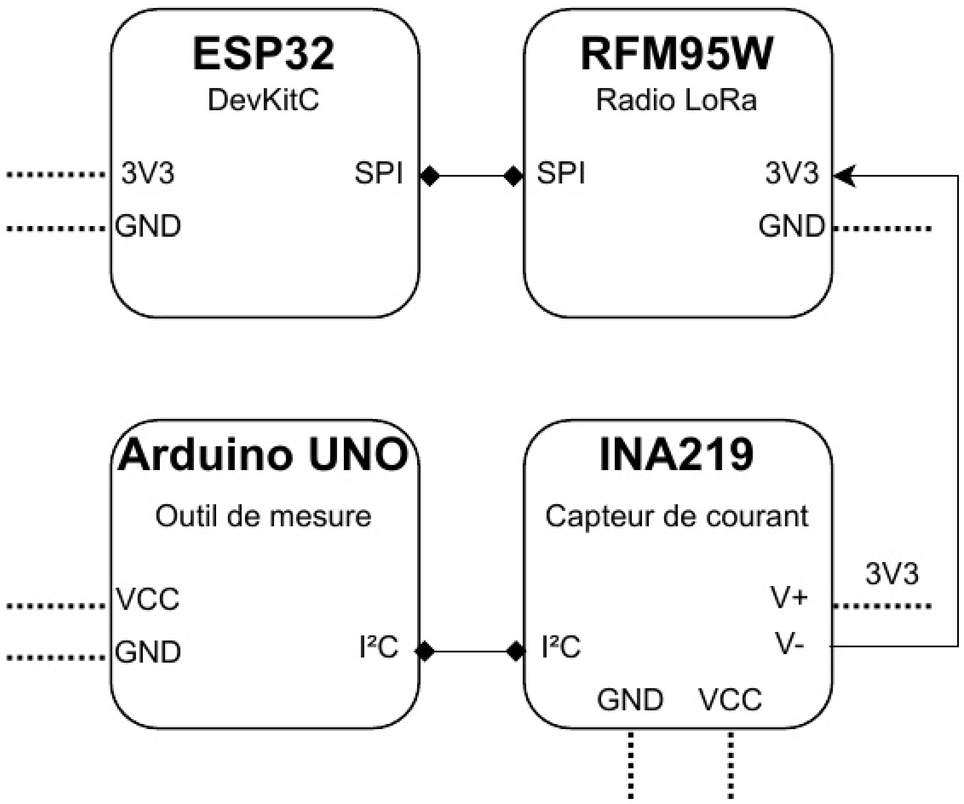


Figure – Représentation de la structure de notre implémentation.

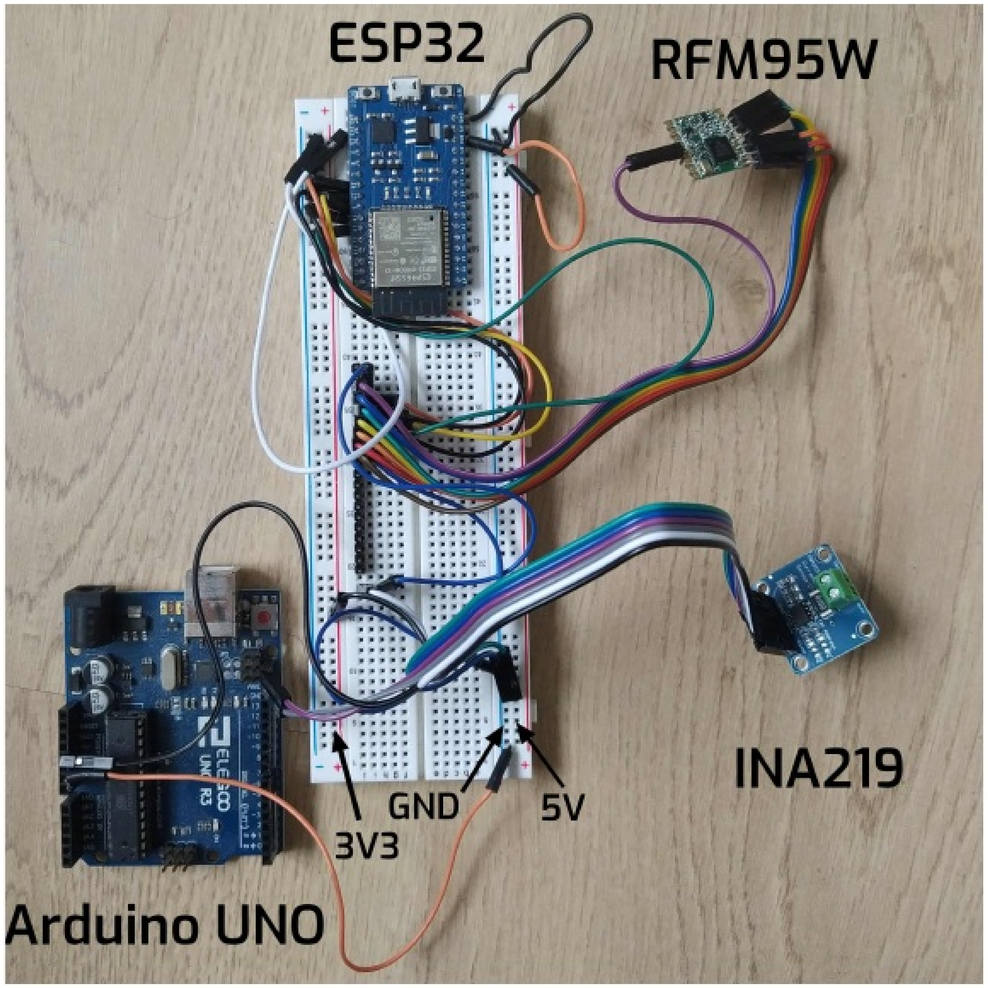
Phase d'expérimentation

Plusieurs idées :

1. Déterminer les zones accessibles selon la puissance d'émission
2. Consommation des modules

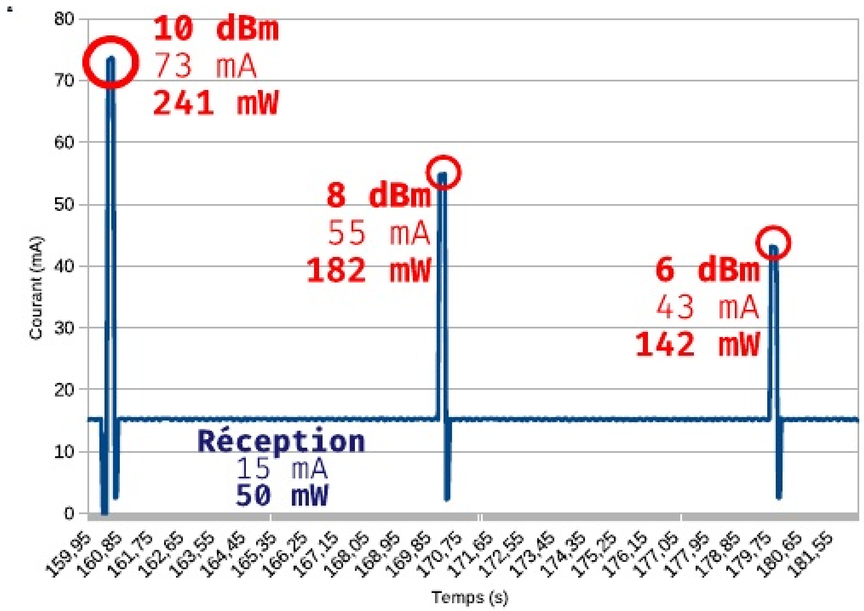


(a) Schéma simplifié

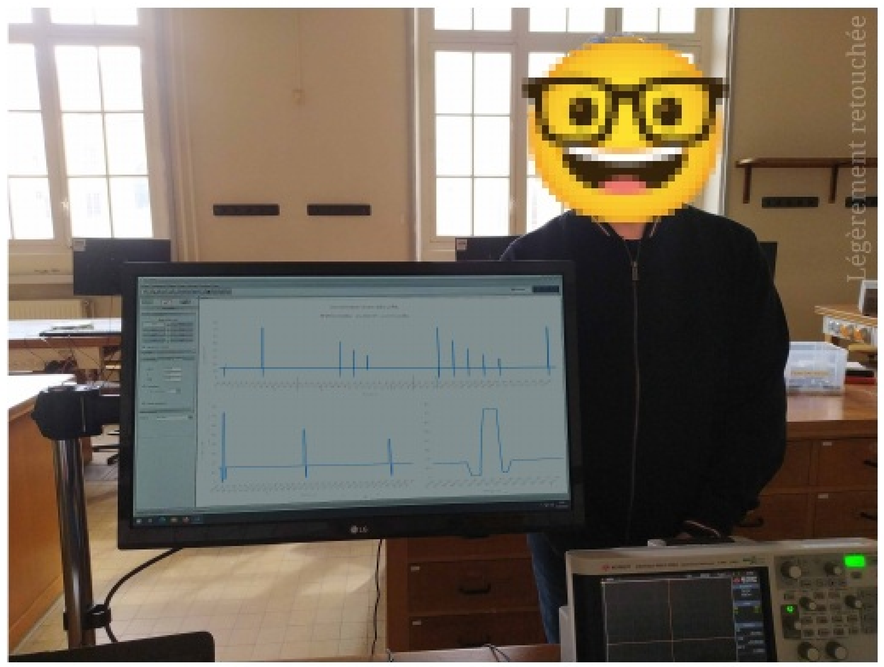


(b) Montage

Mesures



(a) Consommation (assimilée au courant) d'un module LoRa



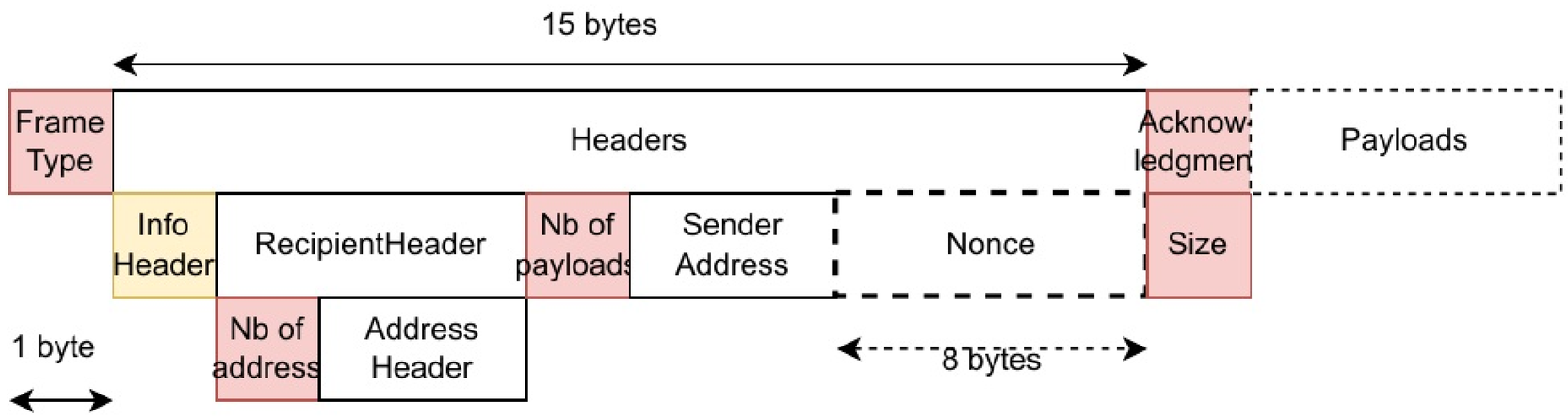
(b) Photo lors de la prise des mesures

1. Résultats conformes aux données constructeurs.
2. Semble indiquer que nos efforts sont négligeables.

Conclusion

1. Difficultés à évaluer les performances du modèle
   1. Consommation majoritairement dûe à l'écoute active
   2. Des techniques avec un intéret variable selon le type d'appareil
2. Dépendance importante au fonctionnement global de l'objet

   Hibernation, Fréquence d'envoi, Réception programmée,...
3. Cependant un surcoût faible
   1. En ignorant les avantages du protocole, surcoût de seulement 5 octets.



2. Une intéropérabilité néanmoins avantageuse

Conclusion

1. Adapté à des passerelles mobiles (Mobile gateways)
2. Dans un modèle classique : Client - Gateway
    1. Surcoût d'au plus 5 octets pour les clients.
    2. Distribution efficace de paquets pour la passerelle fixe
    Économe en énergie, Adapté au grand nombre d'échanges
3. Application concrète : Amazon Sidewalk

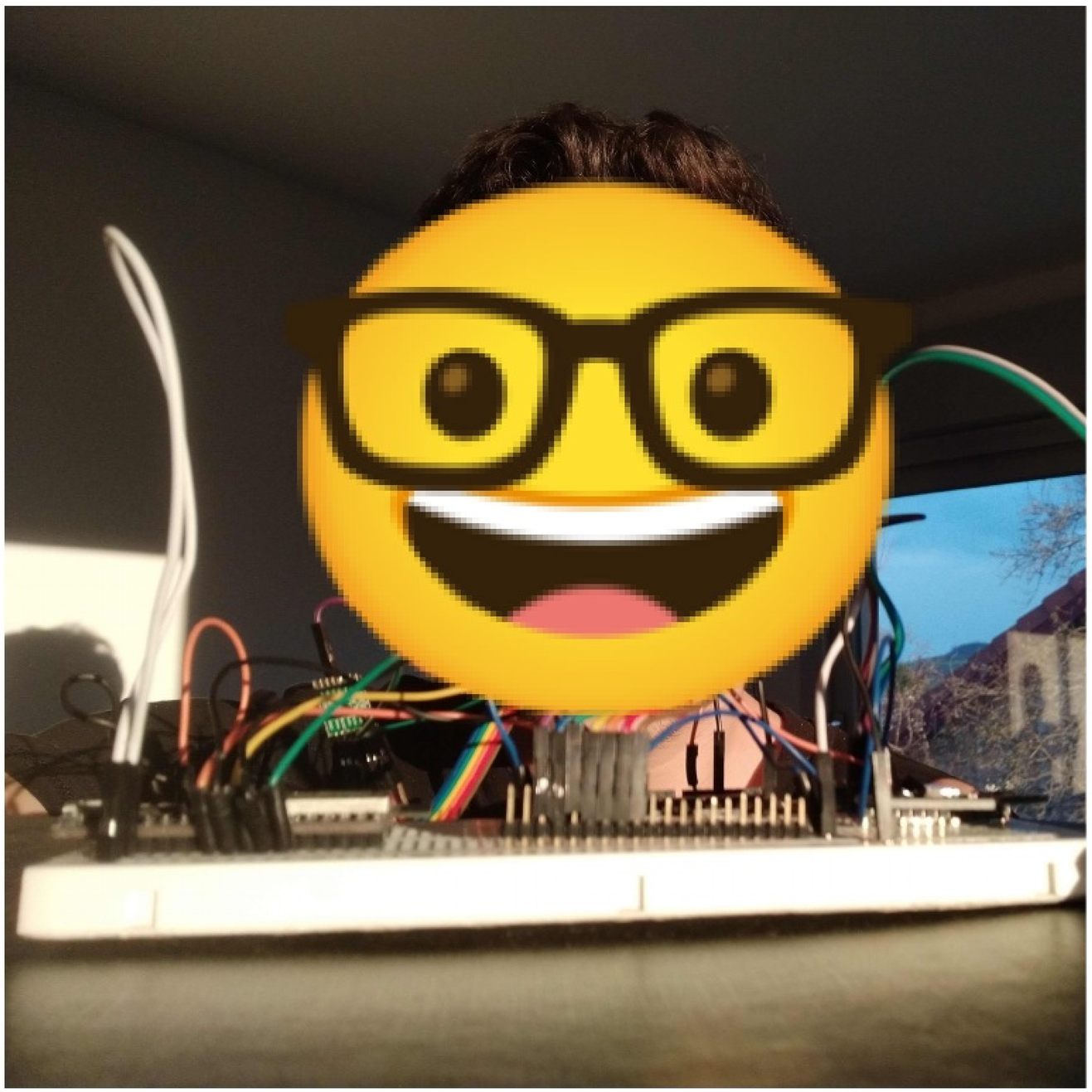
Ouverture

Quelques pistes pour la suite :

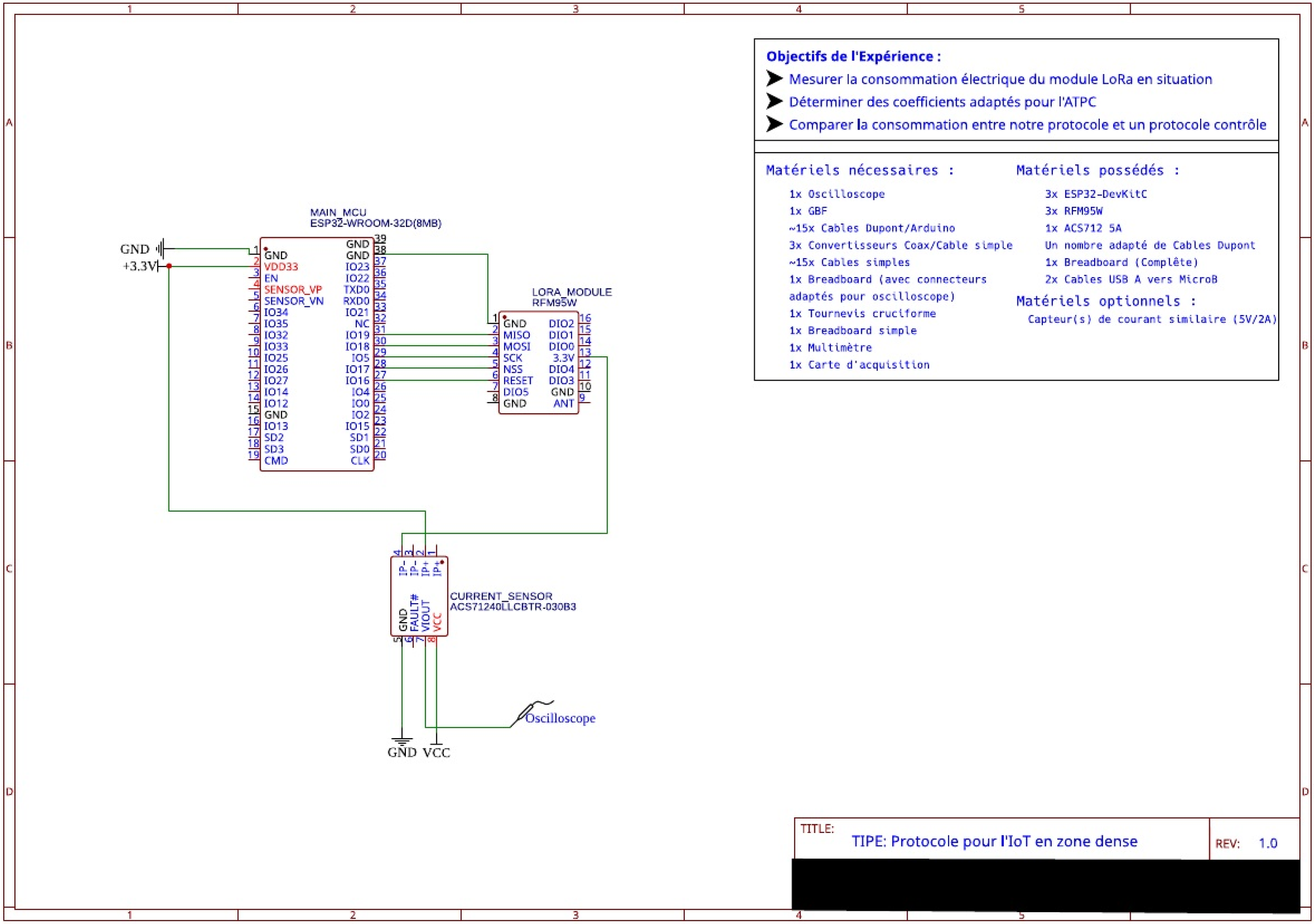
1. Critiques
    1. Déni de service
    2. Sécurité des échanges
2. Négociation des plages d'hibernation au niveau du lien
3. Utilisation d'un Coprocesseur Ultra Faible Puissance

Fin

Merci.



Annexe A - Schéma Montage 1 (ACS712)



Objectifs de l'Expérience :

- Mesurer la consommation électrique du module LoRa en situation
- Déterminer des coefficients adaptés pour l'ATPC
- Comparer la consommation entre notre protocole et un protocole contrôle

Matériels nécessaires :

- 1x Oscilloscope
- 1x GBF
- ~15x Cables Dupont/Arduino
- 3x Convertisseurs Coax/Cable simple
- ~15x Cables simples
- 1x Breadboard (avec connecteurs adaptés pour oscilloscope)
- 1x Tournevis cruciforme
- 1x Breadboard simple
- 1x Multimètre
- 1x Carte d'acquisition

Matériels possédés :

- 3x ESP32-DevKitC
- 3x RFM95W
- 1x ACS712 5A
- Un nombre adapté de Cables Dupont
- 1x Breadboard (Complète)
- 2x Cables USB A vers MicroB

Matériels optionnels :

- Capteur(s) de courant similaire (5V/2A)

TITLE: TIPE: Protocole pour l'IoT en zone dense

REV: 1.0

[redacted]

Annexe B1 - Consommation Module LoRa

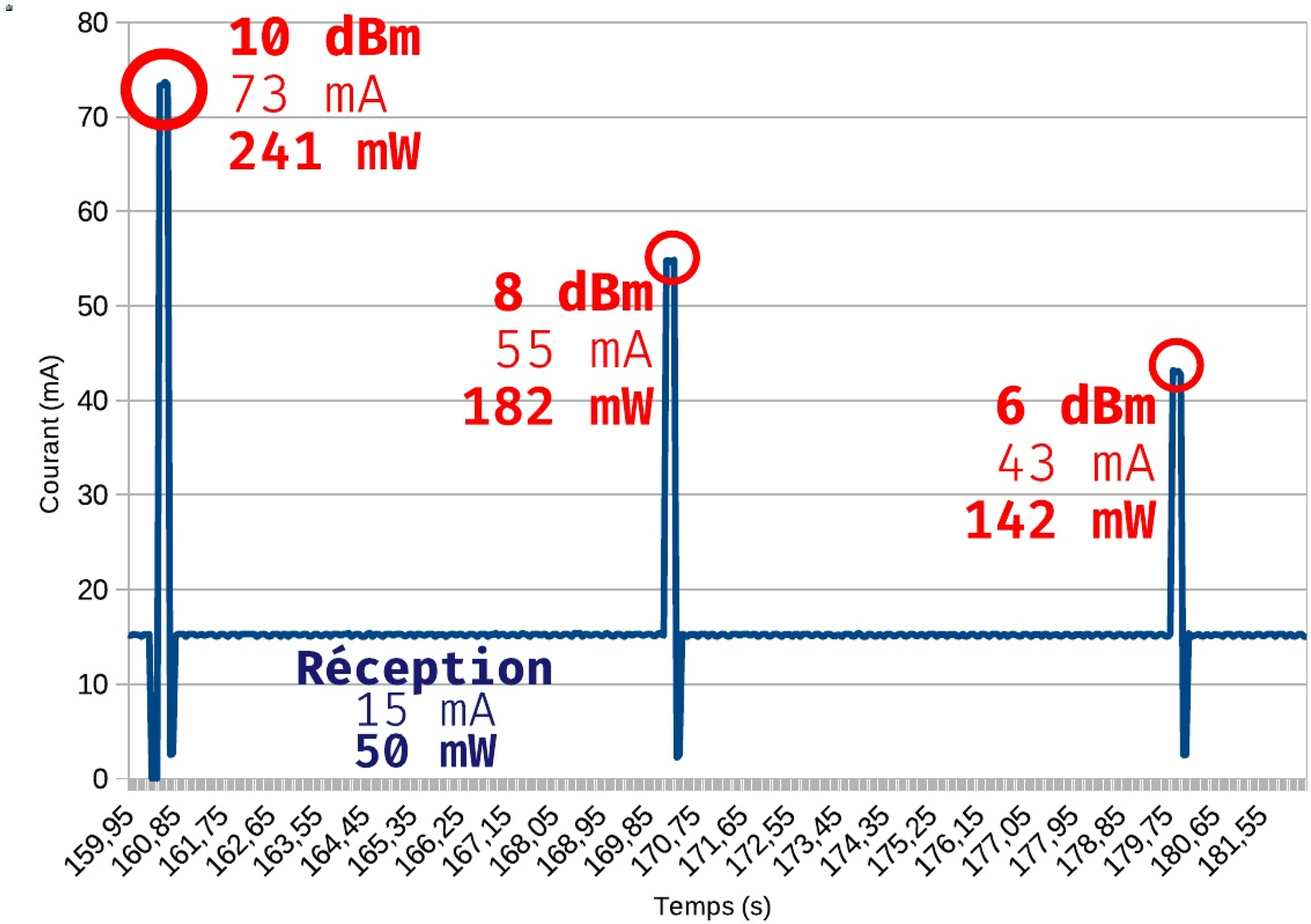


Figure – Consommation (assimilée au courant) d'un module LoRa

Annexe B2 - Consommation Module LoRa

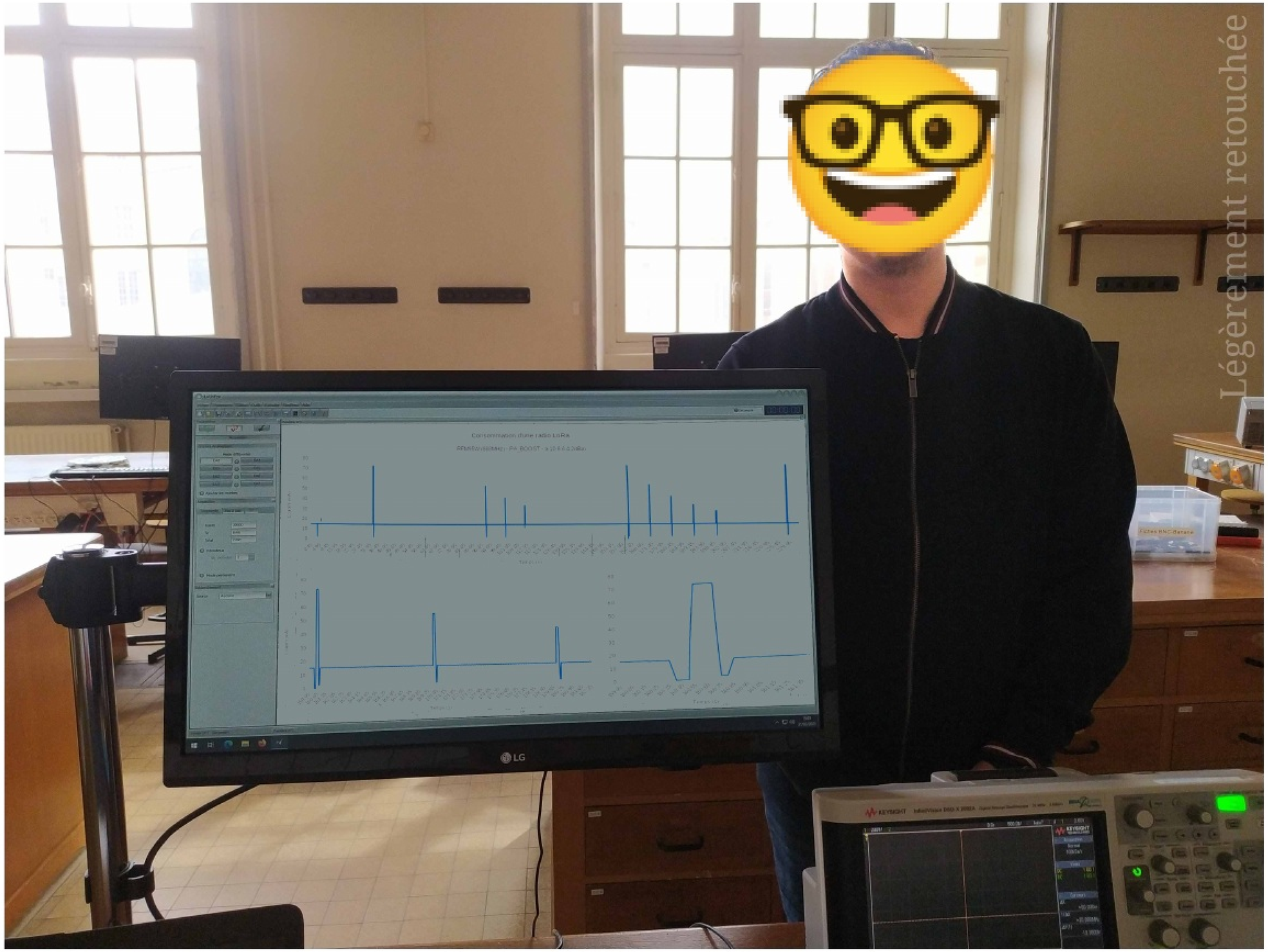
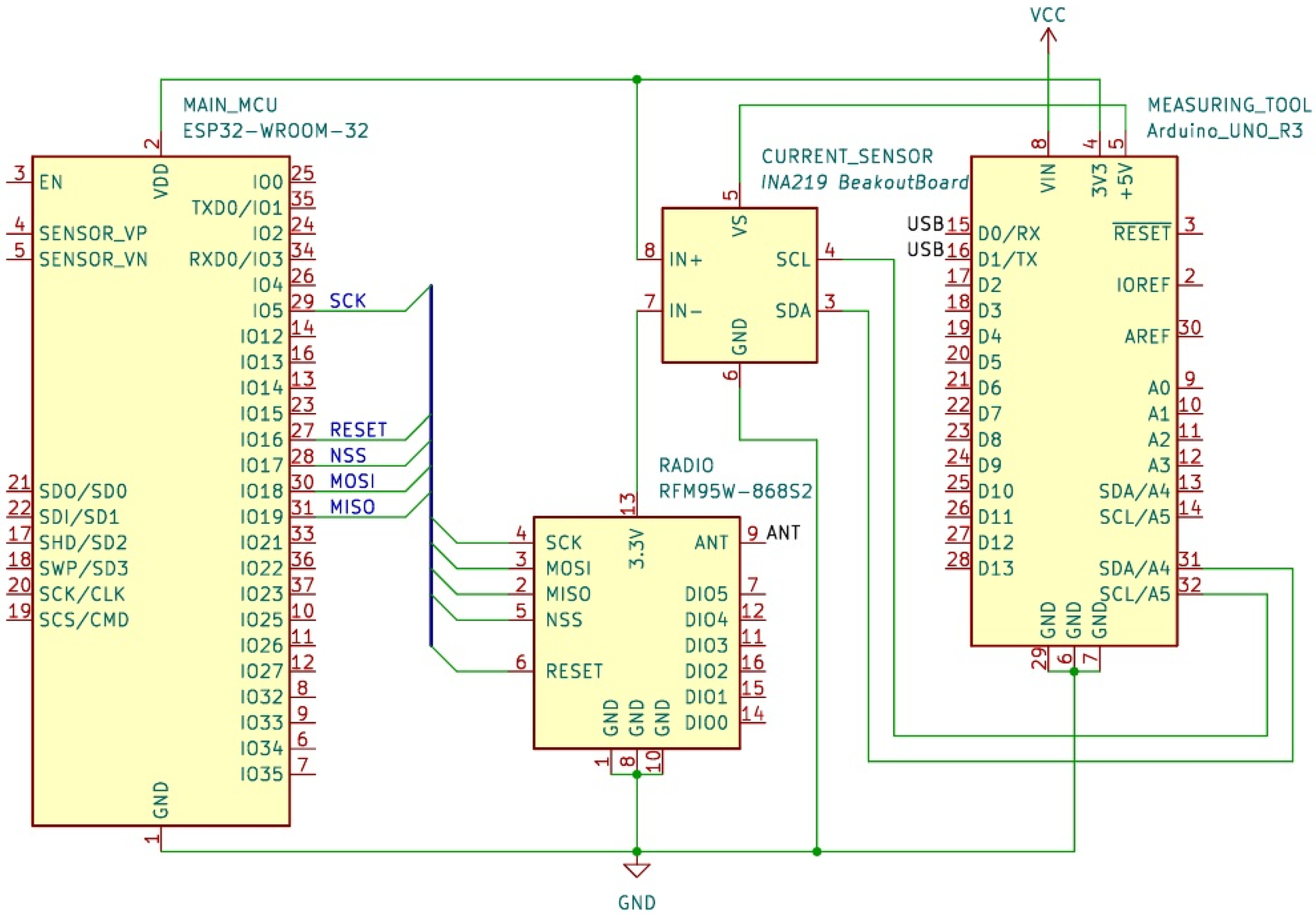


Figure – Photo lors de la prise des mesures

Annexe B3 - Consommation Module LoRa



Annexe C - Bibliothèque

```rust
/// Radio Device trait, representation for a specific device implementing the protocol.
pub trait Device<'a> {
    type DeviceError;

    /// Get transmission status.
    fn is_transmitting(&mut self) -> Result<bool, Self::DeviceError>;

    /// Get listening status.
    fn is_listening(&mut self) -> Result<bool, Self::DeviceError>;

    /// Flush the packet queue and transmit it using its current state.
    fn transmit(&mut self) -> Result<FrameNonce, Self::DeviceError>;

    /// Put the device in listening mode, waiting to receive new packets on its address.
    ///
    /// Periodical check need to be made with [[Device::check_reception]] to poll internal radio state
    /// and retrieve the received message by the physical device.
    fn start_reception(&mut self) -> Result<(), Self::DeviceError>;

    /// Check reception of messages by the physical radio.
    ///
    /// Periodical check need to be made with this method to poll internal radio state
    /// and retrieve the received message by the physical device.
    fn check_reception(&mut self) -> Result<bool, Self::DeviceError>;

    /// Queue and prepare acknowledgements (due to a successful reception) for the next frame.
    fn queue_acknowledgements(
        &mut self,
    ) -> Result<bool, QueueError<Self::DeviceError>>;

    /// Add given payload as packet to the internal queue.
    fn queue<'b>(
        &mut self,
        dest: LoRaDestination,
        payload: &'b [u8],
        ack: bool,
    ) -> Result<(), QueueError<Self::DeviceError>>;

  //...
}
```

Annexe D1 - INA219 – Functional Block Diagram

8.2 Functional Block Diagram

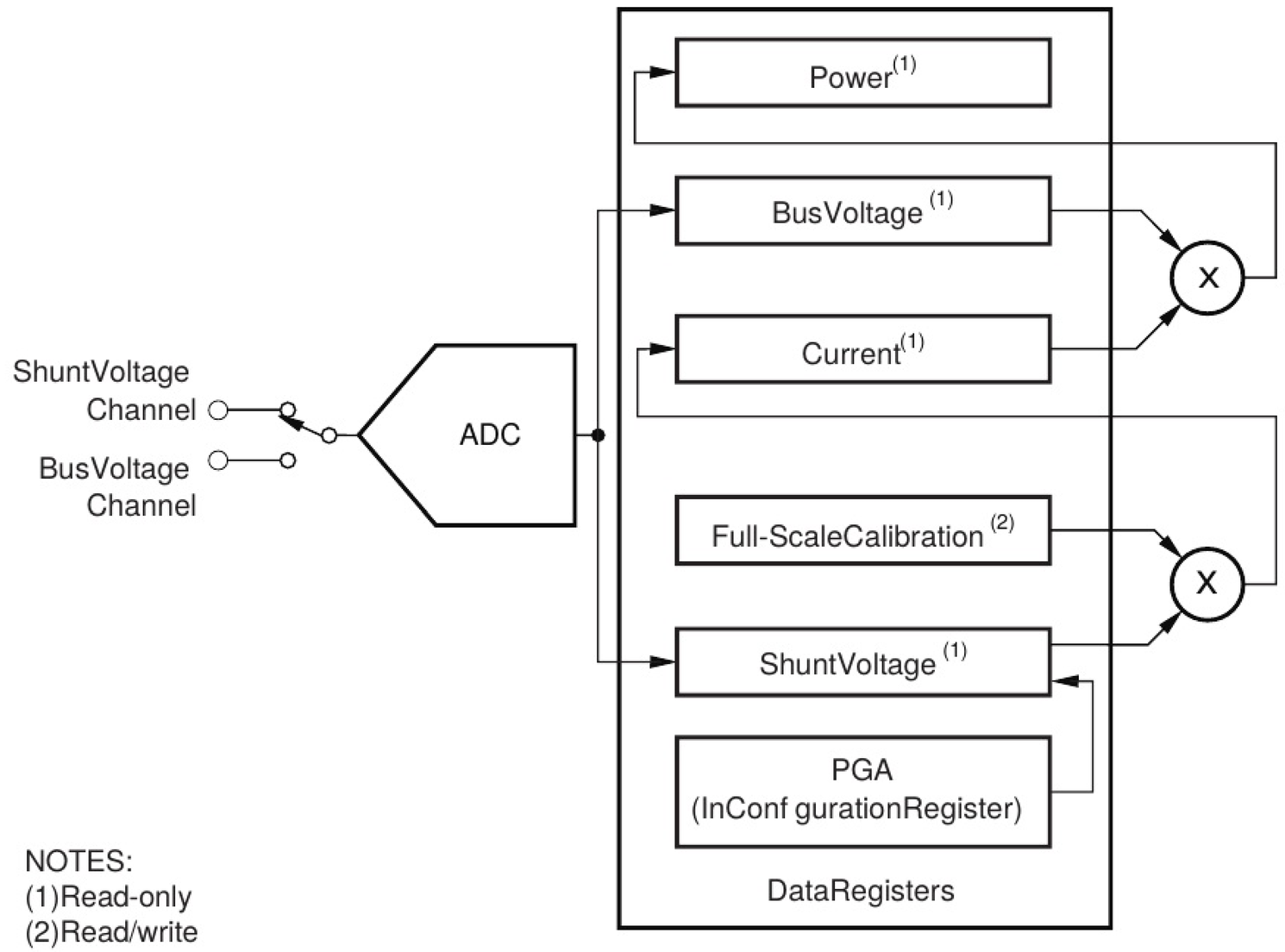


Figure – Extrait de la spécification technique des modules INA219 de Texas Instruments (SBOS448G)

Feature Description (continued)

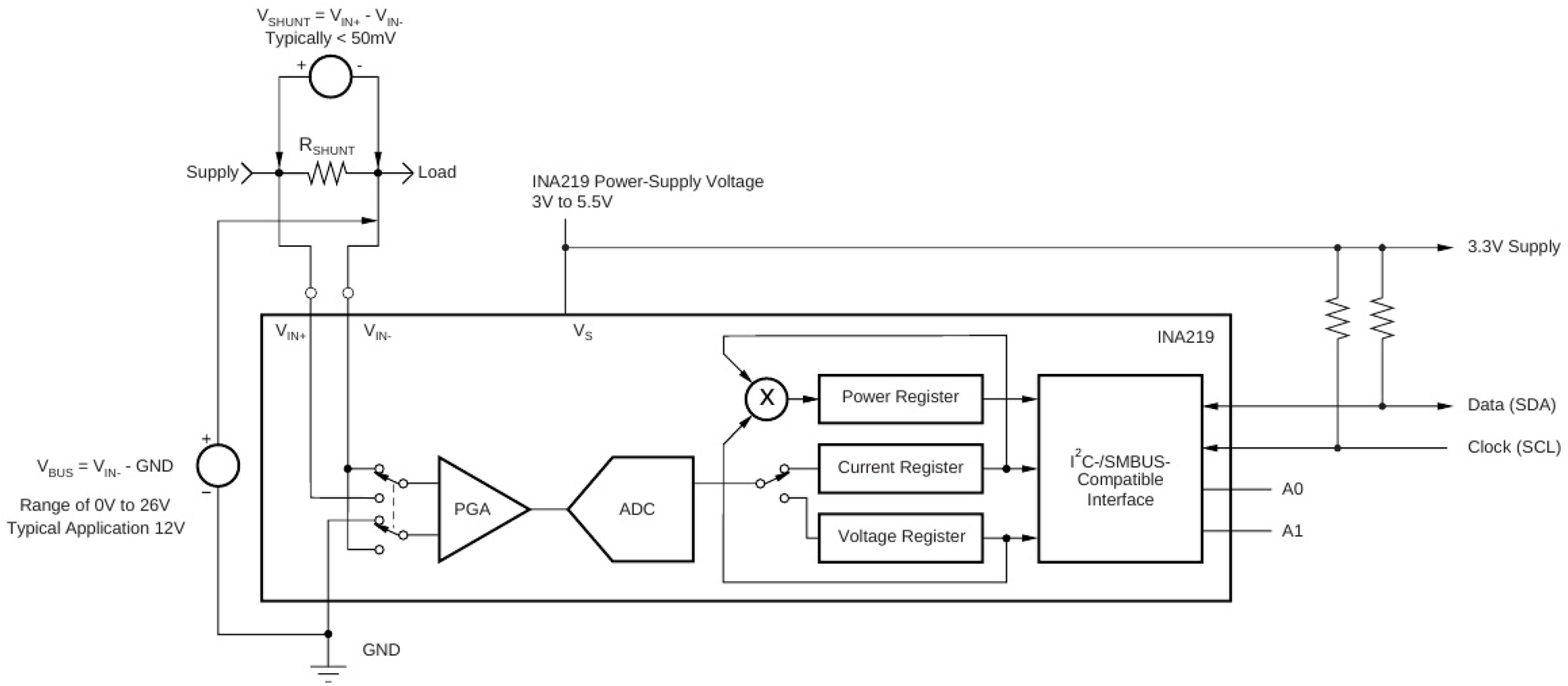


Figure 13. INA219 Configured for Shunt and Bus Voltage Measurement

Figure – Extrait de la spécification technique des modules INA219 de Texas Instruments (SBOS448G)

Annexe LO - Listings I

Récapitulatif des listings disponibles :

Annexe L1 - radio-tipe-poc I

Listing – radio-tipe-poc/Cargo.toml

```
1  [package]
2  name = "radio-tipe-poc"
3  version = "0.1.0"
4  edition = "2021"
5
6  # See more keys and their definitions at https://doc.rust-lang.org/cargo/
       reference/manifest.html
7
8  [dependencies]
9  radio = { path = "../radio-hal" }
10 embedded-hal = "0.2"
11 thiserror = "1"
12 smol = "1.2"
13 radio-sx127x = { path = "../rust-radio-sx127x" }
14 serde = { version = "1.0", features = ["derive"] }
15 log = "*"
16 ringbuf = "0.3"
17 lru = "0.10"
18 getrandom = "0.2.9"
```

Annexe L1 - radio-tipe-poc II

Listing – radio-tipe-poc/src/lib.rs

```
1  //! # Radio TIPE PoC
2  //!
3  //! This library is the central piece of a TIPE (academic project), and should
       allow anybody
4  //! to use this protocol to exchange messages and replicate our results with
       similar hardware.
5  //!
6  //! ## Goals
7  //! - Provide a real implementation of this protocol that has been proposed.
8  //! - Provide an implementation that works on embedded devices like the ESP32-
       DevKitC
9  //! - Provide a library for application uses. This ensures we have properly
       structure our network
10 //!   for real use cases.
11 //!
12 //! ## Considerations
13 //! - This library has only been tested on ESP32-DevKitC and RFM95W modules.
14 //! - This library relies lightly on ‘rust-radio-sx127x‘, therefore you will
       need
15 //!   a LoRa radio based on the SX127x radio.
16 //! - This library uses the standard library, something that might not be
       available on most
17 //!   embedded platforms.
18 //!
19 //! ## Caution
20 //!
21 //! Please note that this project is an academic/research project and will make
22 //! some assumptions on the hardware and the actual frames received by the
       physical
```

Annexe L1 - radio-tipe-poc III

```rust
//! radio. DO NOT USE THIS PROJECT FOR REAL USES. It does not enforce any security
//! and will not enforce authenticity neither integrity of the communication.
//!
//! ## Usage
//! Some examples are available at modules [crate::device] and [crate::radio].

pub mod atpc;
pub mod device;
pub mod frame;
pub mod radio;

/// Representation of the recipients for a particular message that will be
/// send or has been received by the LoRa radio.
pub enum LoRaDestination {
    /// This message is for everyone listening.
    ///
    /// Similar to the concept of broadcast in the LAN/WAN world.
    Global,
    /// This message is intended for a group of peers.
    Group(Vec<LoRaAddress>),
    /// This message is intended for a single peer of the network.
    Unique(LoRaAddress),
}

/// Simple alias for the representation of a peer address.
///
/// Some might be more familiar with the similar MAC addresses. Indeed it actually
/// is the physical name of the device and only helps establish link-to-link
/// transmissions.
```

Annexe L1 - radio-tipe-poc IV

```rust
pub type LoRaAddress = u16;
```

Listing – radio-tipe-poc/src/atpc.rs

```rust
//! Adaptive Transmission Power Control interfaces and basic implementations.
//!
//! This module provides the public trait to implement an ATPC at the application level.
//! Moreover it provides two implementations, a naive implementation that basically disable
//! the ATPC and a [standard implementation](DefaultATPC) based on
//! [Shan Lin's work](https://www.cs.virginia.edu/~stankovic/psfiles/ATPC.pdf).
//!
//! ## Usages
//! Either just use a provided implementation and passed it to your [LoRaRadio](crate::radio::LoRaRadio).
//! ```rust,ignore
//! let atpc = radio_tipe_poc::atpc::TestingATPC::new(vec![10, 8, 6, 4, 2]);
//! let mut device = LoRaRadio::new(lora, &channels, atpc, -100, None, None, 0b0101_0011);
//! ```
//! Or implement your own ATPC by creating your structure who implement the [ATPC] trait.
use crate::frame::FrameNonce;
use crate::LoRaAddress;

use std::cmp::Ordering;
use std::num::NonZeroUsize;
use std::time::Duration;
```

Annexe L1 - radio-tipe-poc V

```rust
use std::time::Instant;

use lru::LruCache;

/// Modelisation of the RSSI on the receiver end when the transmitter uses a particular
/// Transmission Power (Transmission Level).
///
/// This model uses the following approximation: `RSSI = a * TP + b` for a particular `ControlModel(a,b)`.
///
/// This model follows the design provided in [Shan Lin's work](https://www.cs.virginia.edu/~stankovic/psfiles/ATPC.pdf).
#[derive(Clone, PartialEq, Eq, Debug)]
struct ControlModel(i16, i16);

/// Status of a neighbor for the [DefaultATPC].
#[derive(Clone, PartialEq, Eq, Debug)]
enum NeighborStatus {
    /// This neighbor has not yet answered to our beacons (or partially). We currently have no
    /// information on the transmission power needed for this peer.
    Initializing,
    /// This neighbor has been fully initialized. Its control model is valid. It was successfully built
    /// with the answers from the peer to our beacons.
    Runtime,
}

/// Representation of a peer for the [DefaultATPC].
#[derive(Clone, Debug)]
```

Annexe L1 - radio-tipe-poc VI

```rust
struct NeighborModel {
    /// Address of this peer.
    pub node_address: LoRaAddress,
    /// Status of the peer for the ATPC.
    pub status: NeighborStatus,
    /// Dedicated control model for this particular node.
    pub control_model: ControlModel,
    /// RSSI responses for the various transmissions power levels.
    ///
    /// Those are calculated with the acknowledgments given by the peer. This
     includes
    /// the answers to our beacons.
    pub rssi: Vec<i16>,
}

impl Ord for NeighborModel {
    fn cmp(&self, other: &Self) -> Ordering {
        self.node_address.cmp(&other.node_address)
    }
}

impl PartialEq for NeighborModel {
    fn eq(&self, other: &Self) -> bool {
        self.node_address == other.node_address
    }
}
impl Eq for NeighborModel {}

impl PartialOrd for NeighborModel {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(&other))
```

Annexe L1 - radio-tipe-poc VII

```
77      }
78  }
79
80  impl NeighborModel {
81      /// Constructs a new instance of a neighbor model.
82      ///
83      /// Due to its implementation being separated from the [DefaultATPC],
84      /// we need to pass the number of transmission power levels that are
85      /// tracked by the ATPC.
86      fn new(node_address: LoRaAddress, ntp: usize) -> Self {
87          NeighborModel {
88              node_address,
89              status: NeighborStatus::Initializing,
90              control_model: ControlModel(0, 0),
91              rssi: vec![0; ntp],
92          }
93      }
94  }
95
96  /// Abstract representation of an Adaptable Transmission Power Control (ATPC).
97  ///
98  /// This trait is an essential component of the [LoRaRadio](crate::device::radio
        ::LoRaRadio).
99  /// This is this module who determine for each peer the needed transmission
        power to successfully
100 /// transmit a frame to a neighbor while helping reducing the energy consumption
        due to radio
101 /// transmission.
102 pub trait ATPC {
103     /// Should the radio transmit beacons ? It is mostly determined by the time
        elapsed from the last
```

Annexe L1 - radio-tipe-poc VIII

```rust
    /// transmission of beacons and the registration of unknown peers that are waiting for initialization.
    fn is_beacon_needed(&self) -> bool;

    /// Gives a list of transmission power to use to transmit the beacons.
    /// Those might or not be equal to the transmission powers given at construction of an ATPC.
    ///
    /// Note that this function might return an empty Vec if the ATPC does not implement beacon.
    fn get_beacon_powers(&self) -> Vec<i8>;

    /// Registers a beacon with its transmission power (index in the [get_beacon_powers](ATPC::get_beacon_powers))
    /// and its nonce.
    ///
    /// This ensures [report_successful_reception](ATPC::report_successful_reception) can correctly
    /// update the [ControlModel] of each neighbor.
    fn register_beacon(&mut self, tpi: usize, nonce: FrameNonce);

    /// Registers a neighbor. This indicates an interest by the radio to transmit data to this peer.
    ///
    /// This function might cause (if the peer is unknown) a transmission of beacons.
    fn register_neighbor(&mut self, neighbor_addr: LoRaAddress) -> bool;

    /// Unregisters a neighbor. It might force to forget this particular neighbor.
    fn unregister_neighbor(&mut self, neighbor_addr: LoRaAddress) -> bool;
```

Annexe L1 - radio-tipe-poc IX

```
127
128     /// Calculates the needed transmission power for a particular neighbor.
129     fn get_tx_power(&mut self, neighbor_addr: LoRaAddress) -> i8;
130
131     /// Calculates the needed transmission power for a particular set of
          neighbors.
132     fn get_min_tx_power(&mut self, mut neighbor_addrs: Vec<LoRaAddress>) -> (i8,
          Vec<LoRaAddress>) {
133         // Minimal default implementation.
134         let mut tx_power = 0;
135         let mut should_update = Vec::new();
136         neighbor_addrs.sort();
137         for na in &neighbor_addrs {
138             let tp = self.get_tx_power(*na);
139             if tp > tx_power {
140                 tx_power = tp;
141                 should_update.clear();
142                 should_update.push(*na);
143             } else if tp == tx_power {
144                 should_update.push(*na);
145             }
146         }
147         if should_update.len() > 0 {
148             return (tx_power, should_update);
149         } else {
150             return (0, neighbor_addrs);
151         }
152     }
153
154     /// Reports the reception of an acknownledgment (maybe for a beacon) by a
          neighbor.
```

Annexe L1 - radio-tipe-poc X

```
155     ///
156     /// This will update the [ControlModel] of this particular peer accordingly
         to the given
157     /// `drssi` (Delta between the RSSI target and the received RSSI of this
         tranmission).
158     fn report_successful_reception(
159         &mut self,
160         neighbor_addr: LoRaAddress,
161         nonce: FrameNonce,
162         drssi: i16,
163     );
164
165     /// Reports the lack of acknownledgment (maybe for a beacon) by a neighbor.
166     ///
167     /// This will update the [ControlModel] of this particular peer accordingly
168     fn report_failed_reception(&mut self, neighbor_addr: LoRaAddress);
169 }
170
171 /// Default implementation of the ATPC, based on [Shan Lin's work](https://www.
         cs.virginia.edu/~stankovic/psfiles/ATPC.pdf).
172 ///
173 /// It provides an efficient implementation that can adapt to its surrounding
         and with a small cost
174 /// of only three beacon tranmissions per day. Moreover the design is pretty
         simple and offer
175 /// good results in different real case scenarios.
176 pub struct DefaultATPC {
177     /// LRU Cache to remember the parameters associated with the most recent
         neighbors.
178     neighbors: LruCache<LoRaAddress, NeighborModel>,
179     /// The transmission powers usable by the ATPC (and the radio).
```

Annexe L1 - radio-tipe-poc XI

```
180     transmission_powers: Vec<i8>,
181     /// The default transmission power (the index of it in `transmission_powers
          `) that will
182     /// be use if a node is unknown or still initializing.
183     default_tp: u8,
184     /// The minimal RSSI threashold that the radio will consider acceptable.
185     lower_rssi: i16,
186     /// Delay between beacon broadcasting.
187     ///
188     /// 8h seems a good value.
189     beacon_delay: Duration,
190     /// The latest beacons transmitted as a nonce-transmission power level value
          .
191     beacons: LruCache<FrameNonce, u8>,
192     /// Last time a beacon was transmitted.
193     last_beacon: Instant,
194 }
195
196 impl DefaultATPC {
197     /// Builds a new instance of the Default ATPC.
198     pub fn new(
199         transmission_powers: Vec<i8>,
200         default_tp: impl Into<u8>,
201         lower_rssi: i16,
202         beacon_delay: Duration,
203     ) -> Self {
204         let default_tp_ = default_tp.into();
205         let tp_len = transmission_powers.len();
206         assert!(default_tp_ < tp_len as u8);
207         Self {
208             neighbors: LruCache::new(NonZeroUsize::new(128).unwrap()),
```

Annexe L1 - radio-tipe-poc XII

```
209                 transmission_powers,
210                 default_tp: default_tp_,
211                 lower_rssi,
212                 beacons: LruCache::new(NonZeroUsize::new(tp_len + 1).unwrap()),
213                 last_beacon: Instant::now(),
214                 beacon_delay,
215             }
216         }
217
218         /// Rebuilds the [ControlModel] of a specific neighbor.
219         ///
220         /// Mostly used to update a node following a beacon acknowledgment.
221         fn rebuid_neighbor_model(&mut self, neighbor_addr: LoRaAddress) {
222             if let Some(neigh) = self.neighbors.get_mut(&neighbor_addr) {
223                 let n = self.transmission_powers.len();
224                 let sum_tp: f32 = self
225                     .transmission_powers
226                     .iter()
227                     .fold(0.0, |acc, x| acc + (*x as f32));
228                 let sum_rssi: f32 = neigh.rssi.iter().fold(0.0, |acc, x| acc + (*x
    as f32));
229                 let sum_tp_rssi: f32 = (0..self.transmission_powers.len())
230                     .into_iter()
231                     .fold(0.0, |acc, i| {
232                         acc + (self.transmission_powers[i] as f32) * (neigh.rssi[i]
    as f32)
233                     });
234                 let denominator: f32 = (n as f32)
235                     * self
236                         .transmission_powers
237                         .iter()
```

Annexe L1 - radio-tipe-poc XIII

```
238                     .fold(0.0, |acc, x| acc + (*x as f32) * (*x as f32))
239                 + sum_tp * sum_tp;
240
241             neigh.control_model.0 =
242                 (((sum_rssi * sum_tp * sum_tp) - (sum_tp * sum_tp_rssi)) /
     denominator) as i16;
243             neigh.control_model.1 =
244                 ((((n as f32) * sum_tp_rssi) - (sum_tp * sum_rssi)) /
     denominator) as i16;
245             neigh.status = NeighborStatus::Runtime;
246         }
247     }
248
249     /// Updates the [ControlModel] of a specific neighbor.
250     ///
251     /// Mostly used to update a node following a successful/failed transmission.
252     fn update_neighbor_model(&mut self, neighbor_addr: LoRaAddress, delta: i16)
      {
253         let tp = self.get_tx_power(neighbor_addr);
254         if let Some(neigh) = self.neighbors.get_mut(&neighbor_addr) {
255             if (delta > 0 && tp < self.transmission_powers[self.
     transmission_powers.len() - 1])
256                 || (delta < 0 && tp > self.transmission_powers[0])
257             {
258                 neigh.control_model.1 -= delta;
259             }
260         }
261     }
262
263     /// Calculates the transmission power needed for a particular node/neighbor.
264     fn calc_node_tp(&mut self, neighbor_addr: LoRaAddress) -> i8 {
```

Annexe L1 - radio-tipe-poc XIV

```
265         let neigh = self
266             .neighbors
267             .get(&neighbor_addr)
268             .expect("calculating TP for an inexistant neighbor.");
269         let tp_target = (self.lower_rssi - neigh.control_model.1) / neigh.
        control_model.0;
270         if let Some(tp) = self
271             .transmission_powers
272             .iter()
273             .find(|tp| (**tp as i16) >= tp_target)
274         {
275             return *tp;
276         } else {
277             return self.transmission_powers[self.transmission_powers.len() - 1];
278         }
279     }
280 }
281
282 impl ATPC for DefaultATPC {
283     fn is_beacon_needed(&self) -> bool {
284         return self.last_beacon.elapsed() > self.beacon_delay
285             || self
286                 .neighbors
287                 .iter()
288                 .find(|(_, n)| n.status == NeighborStatus::Initializing)
289                 .is_some();
290     }
291
292     fn get_beacon_powers(&self) -> Vec<i8> {
293         return self.transmission_powers.clone();
294     }
```

Annexe L1 - radio-tipe-poc XV

```
295
296     fn register_beacon(&mut self, tpi: usize, nonce: FrameNonce) {
297         self.last_beacon = Instant::now();
298         self.beacons.push(nonce, tpi as u8);
299     }
300
301     fn register_neighbor(&mut self, neighbor_addr: LoRaAddress) -> bool {
302         // We should assure the unicity of the neighbors in the list.
303         if let None = self.neighbors.get(&neighbor_addr) {
304             let neigh = NeighborModel::new(neighbor_addr, self.
      transmission_powers.len());
305             self.neighbors.push(neighbor_addr, neigh);
306             true
307         } else {
308             false
309         }
310     }
311
312     fn unregister_neighbor(&mut self, neighbor_addr: LoRaAddress) -> bool {
313         return self.neighbors.pop_entry(&neighbor_addr).is_some();
314     }
315
316     fn get_tx_power(&mut self, neighbor_addr: LoRaAddress) -> i8 {
317         if self.neighbors.contains(&neighbor_addr) {
318             return self.calc_node_tp(neighbor_addr);
319         }
320         self.transmission_powers[self.default_tp as usize]
321     }
322
323     fn get_min_tx_power(&mut self, mut neighbor_addrs: Vec<LoRaAddress>) -> (i8,
       Vec<LoRaAddress>) {
```

Annexe L1 - radio-tipe-poc XVI

```
324             let mut tx_power = None;
325             let mut should_update = Vec::new();
326             neighbor_addrs.sort();
327             for na in &neighbor_addrs {
328                 let tp = self.get_tx_power(*na);
329                 if tx_power.is_none() || tp == tx_power.unwrap() {
330                     should_update.push(*na);
331                 } else if tp > tx_power.unwrap() {
332                     tx_power = Some(tp);
333                     should_update.clear();
334                     should_update.push(*na);
335                 }
336             }
337             if let Some(tx_power) = tx_power {
338                 (tx_power, should_update)
339             } else {
340                 (
341                     self.transmission_powers[self.default_tp as usize],
342                     neighbor_addrs,
343                 )
344             }
345         }
346
347         fn report_successful_reception(
348             &mut self,
349             neighbor_addr: LoRaAddress,
350             nonce: FrameNonce,
351             drssi: i16,
352         ) {
353             if let Some(tpi) = self.beacons.get(&nonce) {
354                 if let Some(neigh) = self.neighbors.get_mut(&neighbor_addr) {
```

Annexe L1 - radio-tipe-poc XVII

```
355                     neigh.rssi[*tpi as usize] = drssi;
356                     self.rebuid_neighbor_model(neighbor_addr);
357                 }
358             } else {
359                 self.update_neighbor_model(neighbor_addr, drssi);
360             }
361         }
362
363         fn report_failed_reception(&mut self, neighbor_addr: LoRaAddress) {
364             self.update_neighbor_model(neighbor_addr, -30);
365         }
366     }
367
368     /// Testing implementation.
369     ///
370     /// Provides an implementation that cycles all its transmission powers across
          each transmission.
371     /// Moreover it does not implement beacons, and most of its operations are NO-OP
          .
372     pub struct TestingATPC {
373         /// The transmission powers usable by the ATPC (and the radio).
374         transmission_powers: Vec<i8>,
375         /// Literally a counter of each transmission.
376         counter: usize,
377     }
378
379     impl TestingATPC {
380         /// Builds a new instance of a Testing ATPC.
381         pub fn new(transmission_powers: Vec<i8>) -> Self {
382             Self {
383                 transmission_powers,
```

Annexe L1 - radio-tipe-poc XVIII

```
384                 counter: 0,
385             }
386         }
387     }
388
389     impl ATPC for TestingATPC {
390         fn is_beacon_needed(&self) -> bool {
391             false
392         }
393
394         fn get_beacon_powers(&self) -> Vec<i8> {
395             return vec![];
396         }
397
398         fn register_beacon(&mut self, _tpi: usize, _nonce: FrameNonce) {
399             // NO-OP
400         }
401
402         fn register_neighbor(&mut self, _neighbor_addr: LoRaAddress) -> bool {
403             // NO OP
404             true
405         }
406
407         fn unregister_neighbor(&mut self, _neighbor_addr: LoRaAddress) -> bool {
408             // NO OP
409             true
410         }
411
412         fn get_tx_power(&mut self, _neighbor_addr: LoRaAddress) -> i8 {
413             let tp = self.transmission_powers[self.counter];
414             let len = self.transmission_powers.len();
```

```
415             self.counter = (self.counter + 1) % len;
416             return tp;
417         }
418
419         fn get_min_tx_power(&mut self, neighbor_addrs: Vec<LoRaAddress>) -> (i8, Vec
                <LoRaAddress>) {
420             return (self.get_tx_power(*&neighbor_addrs[0]), neighbor_addrs);
421         }
422
423         fn report_successful_reception(
424             &mut self,
425             _neighbor_addr: LoRaAddress,
426             _nonce: FrameNonce,
427             _drssi: i16,
428         ) {
429             // NO OP
430         }
431
432         fn report_failed_reception(&mut self, _neighbor_addr: LoRaAddress) {
433             // NO OP
434         }
435 }
```

Listing – radio-tipe-poc/src/device.rs

```
1  //! Definitions for the abstract device driver.
2  //!
3  //! It is the essential trait that all applications will have to use to interact
       with
4  //! the radio.
5  //!
6  //! ## Usages
7  //!
8  //! Here is a very short example of how to use [Device] to exchange messages.
9  //!
10 //! In most cases, it will run in a infinite loop to poll and push messages to
       the network.
11 //!
12 //! ```rust,ignore
13 //! pub fn spawn(&'a mut self) -> anyhow::Result<()> {
14 //!     // Create a Tx/Rx Client if necessary
15 //!     let handler = ...;
16 //!
17 //!     self.device.set_transmit_client(Box::new(handler.clone()));
18 //!     self.device.set_receive_client(Box::new(handler));
19 //!
20 //!     {
21 //!         use std::sync::mpsc::RecvTimeoutError;
22 //!         let mut should_transmit = false;
23 //!
24 //!         println!("Initializing ATPC (transmitting beacons)...");
25 //!         self.device.start_reception()?;
26 //!         self.device.transmit_beacon()?;
27 //!         self.device.start_reception()?;
```

Annexe L1 - radio-tipe-poc XXI

```
28 //!
29 //!         loop {
30 //!             // Do something that might set should_transmit to true.
31 //!             // Maybe consume message from the Tx/Rx Client?
32 //!
33 //!             // Checks for reception, processes acknowledgment.
34 //!             if self.device.check_reception()? {
35 //!                 println!("We receive a new message :)");
36 //!                 if self.device.queue_acknowledgments()? {
37 //!                     println!("Acknowledging the received messsage.");
38 //!                     should_transmit = true;
39 //!                 }
40 //!             } else {
41 //!                 // When there is a hint that a transmission should happen,
    try to transmit.
42 //!                 if should_transmit {
43 //!                     self.try_transmit()?;
44 //!                     should_transmit = false;
45 //!                     self.device.start_reception()?;
46 //!                 }
47 //!             }
48 //!
49 //!             // When needed, send the initialization beacons.
50 //!             if self.device.is_beacon_needed() {
51 //!                 println!("Transmitting beacons (ATPC Update needed)...");
52 //!                 self.device.transmit_beacon()?;
53 //!                 self.device.start_reception()?;
54 //!             }
55 //!         }
56 //!     }
57 //!     Ok(())
```

```
58 //! }
59 //! ```
60
61 use crate::frame::FrameNonce;
62 use crate::{LoRaAddress, LoRaDestination};
63 use std::sync::Arc;
64
65 /// Wrapper for an error that might be indicated a full queue.
66 #[derive(thiserror::Error, Debug)]
67 pub enum QueueError<T> {
68     /// This error is due to other reasons than a full queue.
69     #[error("Internal device error. Error not linked to queue being full, no
        need to transmit.")]
70     DeviceError(#[from] T),
71     /// This error results from a full queue. The queue must be cleared (by
        transmitting for instance)
72     /// before you call again the function.
73     #[error("Queue is full. Transmit first to clear the queue and try again.")]
74     QueueFullError(#[source] T),
75 }
76
77 /// Device trait represents a unit system that can receive and send messages
        using
78 /// some complex features like Adaptive-Rate-Power-Rate, Acknowledgment or
        Packet Aggregation.
79 ///
80 /// A small example is available at the [module level](crate::device).
81 // TODO: Give default implementation for most of the inner method when they are
        not related to
82 // a specific radio implementation.
83 //
```

Annexe L1 - radio-tipe-poc XXIII

```rust
// TODO: Implement a Mock device using the MockRadio provided by the radio crate.
pub trait Device<'a> {
    type DeviceError;

    /// Register the new transmission client which will recieve packet acknowledgment and
    /// transmission completion signal.
    fn set_transmit_client(&mut self, client: Box<dyn TxClient>);

    /// Register the new reciever client which will be call for every packet received matching
    // the device address.
    fn set_receive_client(&mut self, client: Box<dyn RxClient>);

    /// Register this device with a new address.
    fn set_address(&mut self, address: LoRaAddress);

    /// Retrieve the current registered address for this device.
    fn get_address(&self) -> LoRaAddress;

    /// Get transmission status.
    fn is_transmitting(&mut self) -> Result<bool, Self::DeviceError>;

    /// Get listening status.
    fn is_listening(&mut self) -> Result<bool, Self::DeviceError>;

    /// Flush the packet queue and transmit it using its current state.
    ///
    /// NO-OP if the queue is empty.
    fn transmit(&mut self) -> Result<FrameNonce, Self::DeviceError>;
```

Annexe L1 - radio-tipe-poc XXIV

```
112
113     /// Put the device in listening mode, waiting to recieve new packets on its
          address.
114     ///
115     /// Periodical check need to be made with [Device::check_reception] to poll
          internal radio state
116     /// and retrieve the received message by the physical device.
117     fn start_reception(&mut self) -> Result<(), Self::DeviceError>;
118
119     /// Check reception of messages by the physical radio.
120     ///
121     /// Periodical check need to be made with this method to poll internal radio
            state
122     /// and retrieve the received message by the physical device.
123     ///
124     /// Note that this method can fail if the physical radio is not in reception
            mode (you should use
125     /// [[Device::start_reception]] for that). Ypu might check this mode by
          using [Device::is_listening].
126     ///
127     /// Please note that you **MUST** acknowledge successful reception by
          calling [Device::queue_acknowledgments] in the
128     /// 60s following this call. This is not done automatically by design, to
          allow packet aggregation and avoid
129     /// a transmission in a function called "reception".
130     fn check_reception(&mut self) -> Result<bool, Self::DeviceError>;
131
132     /// Queue and prepare acknowledgments (due to a successful reception) for
          the next frame.
133     ///
```

Annexe L1 - radio-tipe-poc XXV

```
134     /// Returns [QueueError], on [QueueError::QueueFullError] queue need to be
         flush and transmit
135     /// before being able to call again this function.
136     fn queue_acknowledgments(&mut self) -> Result<bool, QueueError<Self::
         DeviceError>>;
137
138     /// Add given payload as packet to the internal queue.
139     ///
140     /// Returns [QueueError], on [QueueError::QueueFullError] queue need to be
         flush and transmit
141     /// before appending new packets.
142     fn queue<'b>(
143         &mut self,
144         dest: LoRaDestination,
145         payload: &'b [u8],
146         ack: bool,
147     ) -> Result<(), QueueError<Self::DeviceError>>;
148
149     /// Informs the application that the ATPC/radio would like to send beacons.
150     fn is_beacon_needed(&mut self) -> bool;
151
152     /// Forces the radio to send ATPC beacons.
153     fn transmit_beacon(&mut self) -> Result<(), QueueError<Self::DeviceError>>;
154 }
155
156 /// Transmission client, that acts like a callback on transmission of a message.
157 ///
158 /// Device will call this function to acknowledge completion and/or reception
159 /// of a previously queued payload.
160 pub trait TxClient {
161     /// Device acknowledgment of transmission completed
```

Annexe L1 - radio-tipe-poc XXVI

```
162     fn transmission_done(&self, nonce: FrameNonce) -> Result<(), ()>;
163
164     /// Transmission was successful, got an acknowledgment from the given
         recipient for this particular message.
165     fn transmission_successful(&self, recipient: LoRaAddress, nonce: FrameNonce)
           -> Result<(), ()>;
166
167     /// Transmission failed, while an acknowledgment was required, none was
         received by the device from the given recipient for this
168     /// particular message.
169     /// A retransmission can be asked by using [[Device::queue]] with the passed
           payload.
170     fn transmission_failed(
171         &self,
172         sender: LoRaAddress,
173         nonce: FrameNonce,
174         payload: Vec<u8>,
175     ) -> Result<(), ()>;
176 }
177
178 impl<T> TxClient for Arc<T>
179 where
180     T: TxClient,
181 {
182     fn transmission_done(&self, nonce: FrameNonce) -> Result<(), ()> {
183         return T::transmission_done(self.as_ref(), nonce);
184     }
185
186     fn transmission_successful(&self, recipient: LoRaAddress, nonce: FrameNonce)
           -> Result<(), ()> {
187         return T::transmission_successful(self.as_ref(), recipient, nonce);
```

Annexe L1 - radio-tipe-poc XXVII

```
188         }
189
190         fn transmission_failed(
191             &self,
192             recipient: LoRaAddress,
193             nonce: FrameNonce,
194             payload: Vec<u8>,
195         ) -> Result<(), ()> {
196             return T::transmission_failed(self.as_ref(), recipient, nonce, payload);
197         }
198 }
199
200 /// Reception client, acts like a callback on reception of radio messages.
201 ///
202 /// The inner functiosn will be called when the device will receive new payloads
          .
203 pub trait RxClient {
204     /// Device has received the given message.
205     fn receive(&self, sender: LoRaAddress, payload: Vec<u8>, nonce: FrameNonce)
          -> Result<(), ()>;
206 }
207
208 impl<T> RxClient for Arc<T>
209 where
210     T: RxClient,
211 {
212     fn receive(&self, sender: LoRaAddress, payload: Vec<u8>, nonce: FrameNonce)
          -> Result<(), ()> {
213         return T::receive(self.as_ref(), sender, payload, nonce);
214     }
215 }
```

Annexe L1 - radio-tipe-poc XXVIII