1 Annexes

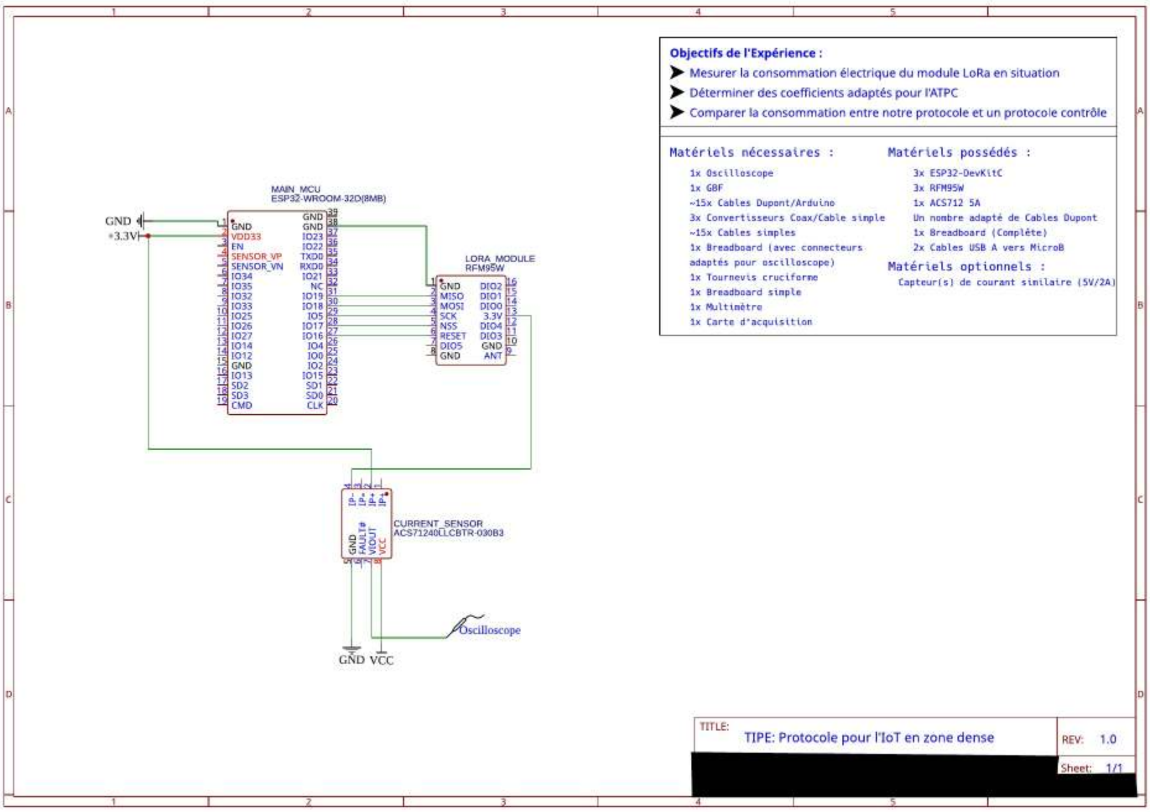
1.1 Sommaire

Récapitulatif des listings disponibles :

Documentation de radio-tipe-poc :

[redacted]

1.2 Annexe A - Schéma Montage 1 (ACS712)



1.3 Annexe B1 - Consommation Module LoRa

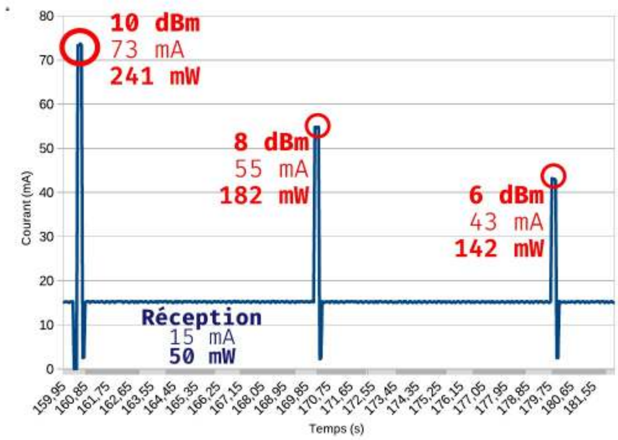
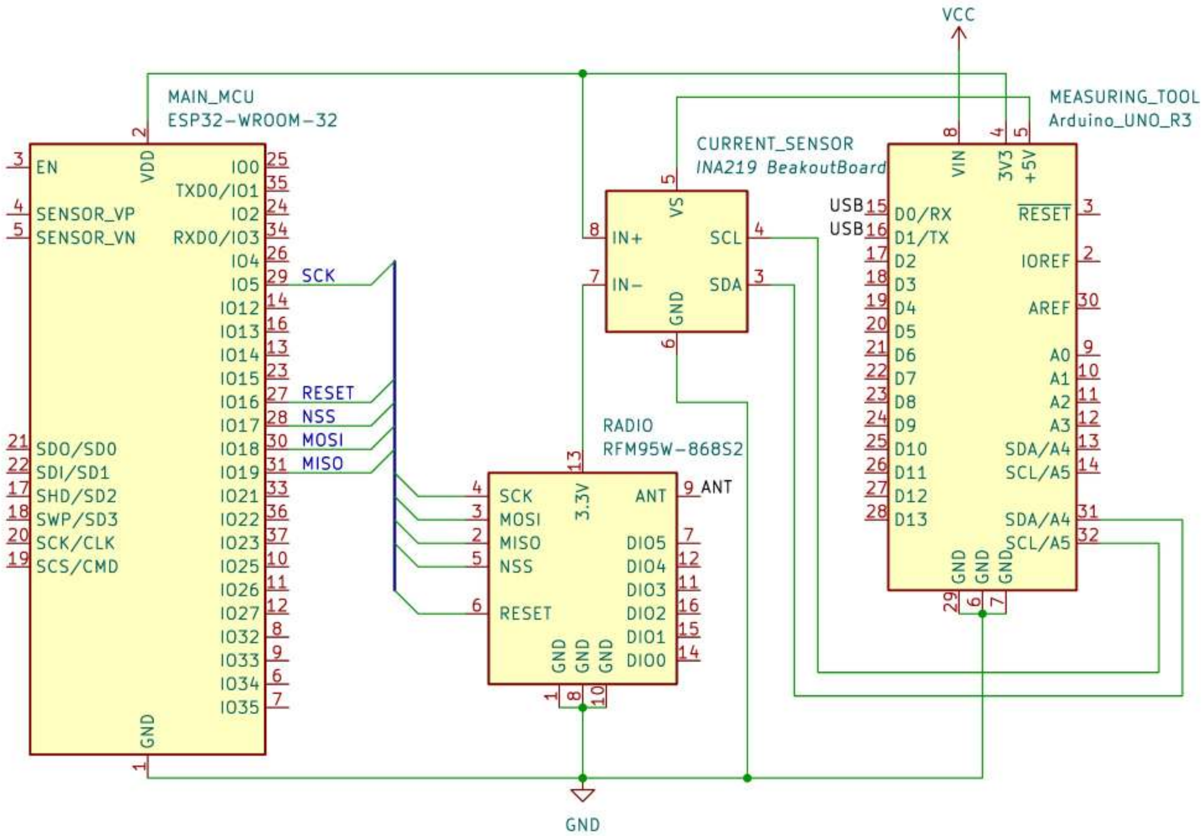


FIGURE 1 – Consommation (assimilée au courant) d'un module LoRa

1.4 Annexe B3 - Consommation Module LoRa



1.5 Annexe D1 - INA219 – Functional Block Diagram

8.2 Functional Block Diagram

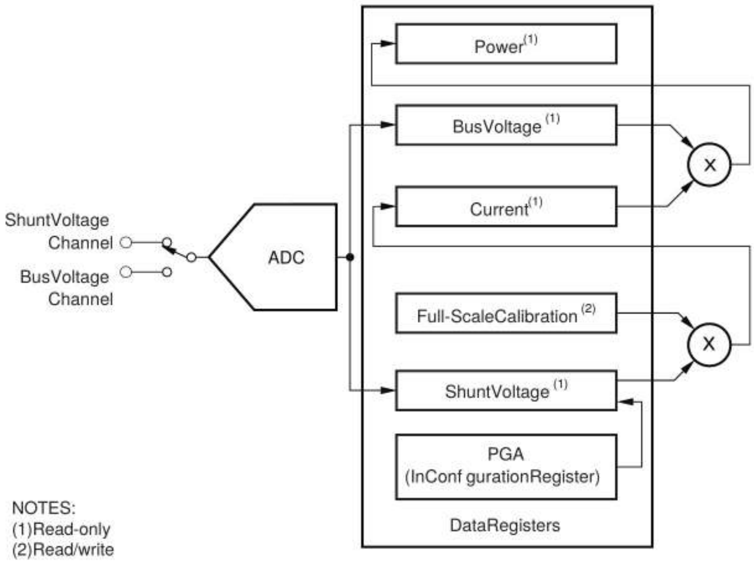


FIGURE 2 – Extrait de la spécification technique des modules INA219 de Texas Instruments (SBOS448G)

1.6 Annexe D2 - INA219 – Technical Schematics

Feature Description (continued)

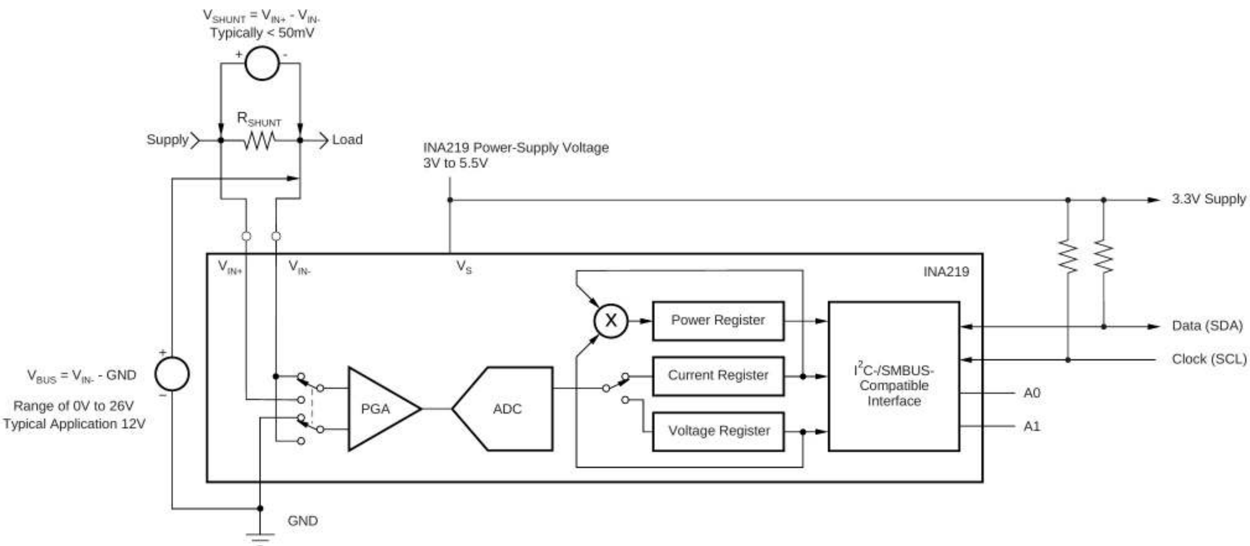


Figure 13. INA219 Configured for Shunt and Bus Voltage Measurement

FIGURE 3 – Extrait de la spécification technique des modules INA219 de Texas Instruments (SBOS448G)

1.7 Annexe D3 - INA219 – Technical Specifications

Les trois prochaines pages sont extraite de la spécification technique des modules INA219 de Texas Instruments (SBOS448G), disponible à l'adresse suivante : `https://www.ti.com/lit/ds/symlink/ina219.pdf`

7 Specifications

7.1 Absolute Maximum Ratings

over operating free-air temperature range (unless otherwise noted)(1)

		MIN	MAX	UNIT
V_S	Supply voltage		6	V
Analog Inputs IN+, IN–	Differential (V_{IN+} – $V_{IN–}$)(2)	–26	26	V
	Common-mode(V_{IN+} + $V_{IN–}$) / 2	-0.3	26	V
SDA		GND – 0.3	6	V
SCL		GND – 0.3	V_S + 0.3	V
Input current into any pin			5	mA
Open-drain digital output current			10	mA
Operating temperature		–40	125	°C
T_J	Junction temperature		150	°C
T_{stg}	Storage temperature	–65	150	°C

(1) Stresses beyond those listed under *Absolute Maximum Ratings* may cause permanent damage to the device. These are stress ratings only, which do not imply functional operation of the device at these or any other conditions beyond those indicated under *Recommended Operating Conditions*. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.
(2) V_{IN+} and $V_{IN–}$ may have a differential voltage of –26 to 26 V; however, the voltage at these pins must not exceed the range –0.3 to 26 V.

7.2 ESD Ratings

			VALUE	UNIT
$V_{(ESD)}$	Electrostatic discharge	Human body model (HBM), per ANSI/ESDA/JEDEC JS-001, all pins(1)	±4000	V
		Charged device model (CDM), per JEDEC specification JESD22-C101, all pins(2)	±750	
		Machine Model (MM)	±200	

(1) JEDEC document JEP155 states that 500-V HBM allows safe manufacturing with a standard ESD control process.
(2) JEDEC document JEP157 states that 250-V CDM allows safe manufacturing with a standard ESD control process.

7.3 Recommended Operating Conditions

over operating free-air temperature range (unless otherwise noted)

	MIN	NOM	MAX	UNIT
V_{CM}		12		V
V_S		3.3		V
T_A	–25		85	ºC

7.4 Thermal Information

THERMAL METRIC(1)		INA219		UNIT
		D (SOIC)	DCN (SOT)	
		8 PINS	8 PINS	
$R_{θJA}$	Junction-to-ambient thermal resistance	111.3	135.4	°C/W
$R_{θJC(top)}$	Junction-to-case (top) thermal resistance	55.9	68.1	°C/W
$R_{θJB}$	Junction-to-board thermal resistance	52	48.9	°C/W
$ψ_{JT}$	Junction-to-top characterization parameter	10.7	9.9	°C/W
$ψ_{JB}$	Junction-to-board characterization parameter	51.5	48.4	°C/W
$R_{θJC(bot)}$	Junction-to-case (bottom) thermal resistance	N/A	N/A	°C/W

(1) For more information about traditional and new thermal metrics, see the *Semiconductor and IC Package Thermal Metrics* application report, SPRA953.

7.5 Electrical Characteristics:

At T_A = 25°C, V_S = 3.3 V, V_{IN+} = 12V, V_{SHUNT} = (V_{IN+} – $V_{IN–}$) = 32 mV, PGA = /1, and BRNG[1] = 1, unless otherwise noted.

PARAMETER		TEST CONDITIONS	INA219A			INA219B			UNIT
			MIN	TYP	MAX	MIN	TYP	MAX	
INPUT									
V_{SHUNT}	Full-scale current sense (input) voltage range	PGA = /1	0		±40	0		±40	mV
		PGA = /2	0		±80	0		±80	mV
		PGA = /4	0		±160	0		±160	mV
		PGA = /8	0		±320	0		±320	mV
	Bus voltage (input voltage) range[2]	BRNG = 1	0		32	0		32	V
		BRNG = 0	0		16	0		16	V
CMRR	Common-mode rejection	V_{IN+} = 0 to 26 V	100	120		100	120		dB
V_{OS}	Offset voltage, RTI[3]	PGA = /1		±10	±100		±10	±50[4]	µV
		PGA = /2		±20	±125		±20	±75[4]	µV
		PGA = /4		±30	±150		±30	±75[4]	µV
		PGA = /8		±40	±200		±40	±100[4]	µV
	vs Temperature	T_A = –25°C to 85°C		0.1			0.1		µV/°C
PSRR	vs Power Supply	V_S = 3 to 5.5 V		10			10		µV/V
	Current sense gain error			±40			±40		m%
	vs Temperature	T_A = –25°C to 85°C		1			1		m%/°C
	IN+ pin input bias current	Active mode		20			20		µA
	IN– pin input bias current || $V_{IN–}$ pin input impedance	Active mode		20 || 320			20 || 320		µA || kΩ
	IN+ pin input leakage[5]	Power-down mode		0.1	±0.5		0.1	±0.5	µA
	IN– pin input leakage[5]	Power-down mode		0.1	±0.5		0.1	±0.5	µA
DC ACCURACY									
	ADC basic resolution			12			12		bits
	Shunt voltage, 1 LSB step size			10			10		µV
	Bus voltage, 1 LSB step size			4			4		mV
	Current measurement error			±0.2%	±0.5%		±0.2%	±0.3%[4]	
	over Temperature	T_A = –25°C to 85°C			±1%			±0.5%[4]	
	Bus voltage measurement error			±0.2%	±0.5%		±0.2%	±0.5%	
	over Temperature	T_A = –25°C to 85°C			±1%			±1%	
	Differential nonlinearity			±0.1			±0.1		LSB
ADC TIMING									
	ADC conversion time	12 bit		532	586		532	586	µs
		11 bit		276	304		276	304	µs
		10 bit		148	163		148	163	µs
		9 bit		84	93		84	93	µs
	Minimum convert input low time		4			4			µs
SMBus									
SMBus timeout[6]				28	35		28	35	ms
DIGITAL INPUTS (SDA as Input, SCL, A0, A1)									
	Input capacitance			3			3		pF
	Leakage input current	0 ≤ V_{IN} ≤ V_S		0.1	1		0.1	1	µA
	V_{IH} input logic level		0.7 (V_S)		6	0.7 (V_S)		6	V
	V_{IL} input logic level		–0.3		0.3 (V_S)	–0.3		0.3 (V_S)	V

(1) BRNG is bit 13 of the Configuration register 00h in Figure 19.
(2) This parameter only expresses the full-scale range of the ADC scaling. In no event should more than 26 V be applied to this device.
(3) Referred-to-input (RTI)
(4) Indicates improved specifications of the INA219B.
(5) Input leakage is positive (current flowing into the pin) for the conditions shown at the top of the table. Negative leakage currents can occur under different input conditions.
(6) SMBus timeout in the INA219 resets the interface any time SCL or SDA is low for over 28 ms.

Electrical Characteristics: (continued)

At T_A = 25°C, V_S = 3.3 V, V_{IN+} = 12V, V_{SHUNT} = (V_{IN+} – $V_{IN–}$) = 32 mV, PGA = /1, and BRNG[(1)] = 1, unless otherwise noted.

PARAMETER	TEST CONDITIONS	INA219A			INA219B			UNIT
		MIN	TYP	MAX	MIN	TYP	MAX	
Hysteresis			500			500		mV
OPEN-DRAIN DIGITAL OUTPUTS (SDA)								
Logic 0 output level	I_{SINK} = 3 mA		0.15	0.4		0.15	0.4	V
High-level output leakage current	V_{OUT} = V_S		0.1	1		0.1	1	µA
POWER SUPPLY								
Operating supply range		3		5.5	3		5.5	V
Quiescent current			0.7	1		0.7	1	mA
Quiescent current, power-down mode			6	15		6	15	µA
Power-on reset threshold			2			2		V

7.6 Bus Timing Diagram Definitions[(1)]

		FAST MODE		HIGH-SPEED MODE		UNIT
		MIN	MAX	MIN	MAX	
$f_{(SCL)}$	SCL operating frequency	0.001	0.4	0.001	2.56	MHz
$t_{(BUF)}$	Bus free time between STOP and START condition	1300		160		ns
$t_{(HDSTA)}$	Hold time after repeated START condition. After this period, the first clock is generated.	600		160		ns
$t_{(SUSTA)}$	Repeated START condition setup time	600		160		ns
$t_{(SUSTO)}$	STOP condition setup time	600		160		ns
$t_{(HDDAT)}$	Data hold time	0	900	0	90	ns
$t_{(SUDAT)}$	Data setup time	100		10		ns
$t_{(LOW)}$	SCL clock LOW period	1300		250		ns
$t_{(HIGH)}$	SCL clock HIGH period	600		60		ns
t_F DA	Data fall time		300		150	ns
t_FCL	Clock fall time		300		40	ns
t_RCL	Clock rise time		300		40	ns
t_RCL	Clock rise time for SCLK ≤ 100kHz		1000			ns

(1) Values based on a statistical analysis of a one-time sample of devices. Minimum and maximum values are not ensured and not production tested.

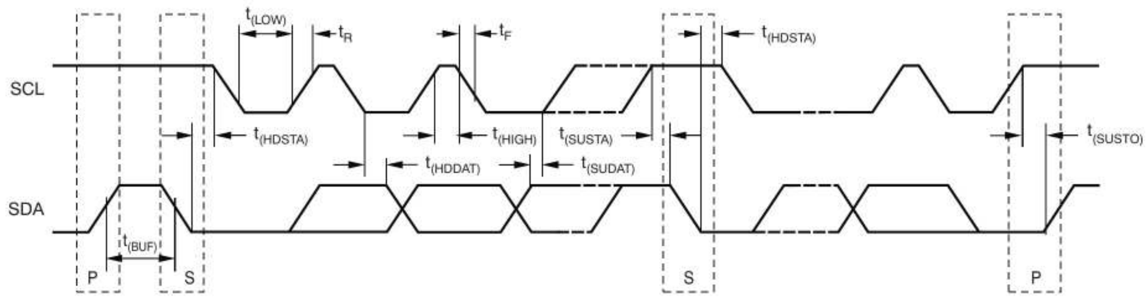


Figure 1. Bus Timing Diagram

2 Listings

2.1 Annexe L1 - radio-tipe-poc

Listing 1 – radio-tipe-poc/Cargo.toml

```toml
[package]
name = "radio-tipe-poc"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
radio = { path = "../radio-hal" }
embedded-hal = "0.2"
thiserror = "1"
smol = "1.2"
radio-sx127x = { path = "../rust-radio-sx127x" }
serde = { version = "1.0", features = ["derive"] }
log = "*"
ringbuf = "0.3"
lru = "0.10"
getrandom = "0.2.9"
```

Listing 2 – radio-tipe-poc/src/lib.rs

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

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

Listing 3 – 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;
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)]
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))
    }
}

impl NeighborModel {
    /// Constructs a new instance of a neighbor model.
    ///
    /// Due to its implementation being separated from the [DefaultATPC],
    /// we need to pass the number of transmission power levels that are
    /// tracked by the ATPC.
    fn new(node_address: LoRaAddress, ntp: usize) -> Self {
        NeighborModel {
            node_address,
            status: NeighborStatus::Initializing,
            control_model: ControlModel(0, 0),
            rssi: vec![0; ntp],
        }
    }
}

/// Abstract representation of an Adaptable Transmission Power Control (ATPC).
///
/// This trait is an essential component of the [LoRaRadio](crate::device::radio::LoRaRadio).
/// This is this module who determine for each peer the needed transmission power to successfully
/// transmit a frame to a neighbor while helping reducing the energy consumption due
```

```
to radio
/// transmission.
pub trait ATPC {
    /// Should the radio transmit beacons ? It is mostly determined by the time elapsed from the last
    /// 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;

    /// Calculates the needed transmission power for a particular neighbor.
    fn get_tx_power(&mut self, neighbor_addr: LoRaAddress) -> i8;

    /// Calculates the needed transmission power for a particular set of neighbors.
    fn get_min_tx_power(&mut self, mut neighbor_addrs: Vec<LoRaAddress>) -> (i8, Vec<LoRaAddress>) {
        // Minimal default implementation.
        let mut tx_power = 0;
        let mut should_update = Vec::new();
        neighbor_addrs.sort();
        for na in &neighbor_addrs {
            let tp = self.get_tx_power(*na);
            if tp > tx_power {
                tx_power = tp;
                should_update.clear();
                should_update.push(*na);
            } else if tp == tx_power {
                should_update.push(*na);
            }
        }
        if should_update.len() > 0 {
            return (tx_power, should_update);
        } else {
            return (0, neighbor_addrs);
        }
    }

    /// Reports the reception of an acknownledgment (maybe for a beacon) by a neighbor.
    ///
    /// This will update the [ControlModel] of this particular peer accordingly to
```

```
    the given
    /// ‘drssi‘ (Delta between the RSSI target and the received RSSI of this tranmission).
    fn report_successful_reception(
        &mut self,
        neighbor_addr: LoRaAddress,
        nonce: FrameNonce,
        drssi: i16,
    );

    /// Reports the lack of acknownledgment (maybe for a beacon) by a neighbor.
    ///
    /// This will update the [ControlModel] of this particular peer accordingly
    fn report_failed_reception(&mut self, neighbor_addr: LoRaAddress);
}

/// Default implementation of the ATPC, based on [Shan Lin's work](https://www.cs.virginia.edu/~stankovic/psfiles/ATPC.pdf).
///
/// It provides an efficient implementation that can adapt to its surrounding and with a small cost
/// of only three beacon tranmissions per day. Moreover the design is pretty simple and offer
/// good results in different real case scenarios.
pub struct DefaultATPC {
    /// LRU Cache to remember the parameters associated with the most recent neighbors.
    neighbors: LruCache<LoRaAddress, NeighborModel>,
    /// The transmission powers usable by the ATPC (and the radio).
    transmission_powers: Vec<i8>,
    /// The default transmission power (the index of it in ‘transmission_powers‘) that will
    /// be use if a node is unknown or still initializing.
    default_tp: u8,
    /// The minimal RSSI threashold that the radio will consider acceptable.
    lower_rssi: i16,
    /// Delay between beacon broadcasting.
    ///
    /// 8h seems a good value.
    beacon_delay: Duration,
    /// The latest beacons transmitted as a nonce-transmission power level value.
    beacons: LruCache<FrameNonce, u8>,
    /// Last time a beacon was transmitted.
    last_beacon: Instant,
}

impl DefaultATPC {
    /// Builds a new instance of the Default ATPC.
    pub fn new(
        transmission_powers: Vec<i8>,
        default_tp: impl Into<u8>,
        lower_rssi: i16,
        beacon_delay: Duration,
    ) -> Self {
        let default_tp_ = default_tp.into();
        let tp_len = transmission_powers.len();
        assert!(default_tp_ < tp_len as u8);
        Self {
            neighbors: LruCache::new(NonZeroUsize::new(128).unwrap()),
            transmission_powers,
            default_tp: default_tp_,
            lower_rssi,
            beacons: LruCache::new(NonZeroUsize::new(tp_len + 1).unwrap()),
            last_beacon: Instant::now(),
            beacon_delay,
        }
```

```
    }

    /// Rebuilds the [ControlModel] of a specific neighbor.
    ///
    /// Mostly used to update a node following a beacon acknowledgment.
    fn rebuid_neighbor_model(&mut self, neighbor_addr: LoRaAddress) {
        if let Some(neigh) = self.neighbors.get_mut(&neighbor_addr) {
            let n = self.transmission_powers.len();
            let sum_tp: f32 = self
                .transmission_powers
                .iter()
                .fold(0.0, |acc, x| acc + (*x as f32));
            let sum_rssi: f32 = neigh.rssi.iter().fold(0.0, |acc, x| acc + (*x as f32));
            let sum_tp_rssi: f32 = (0..self.transmission_powers.len())
                .into_iter()
                .fold(0.0, |acc, i| {
                    acc + (self.transmission_powers[i] as f32) * (neigh.rssi[i] as f32)
                });
            let denominator: f32 = (n as f32)
                * self
                    .transmission_powers
                    .iter()
                    .fold(0.0, |acc, x| acc + (*x as f32) * (*x as f32))
                + sum_tp * sum_tp;

            neigh.control_model.0 =
                (((sum_rssi * sum_tp * sum_tp) - (sum_tp * sum_tp_rssi)) / denominator) as i16;
            neigh.control_model.1 =
                ((((n as f32) * sum_tp_rssi) - (sum_tp * sum_rssi)) / denominator) as i16;
            neigh.status = NeighborStatus::Runtime;
        }
    }

    /// Updates the [ControlModel] of a specific neighbor.
    ///
    /// Mostly used to update a node following a successful/failed transmission.
    fn update_neighbor_model(&mut self, neighbor_addr: LoRaAddress, delta: i16) {
        let tp = self.get_tx_power(neighbor_addr);
        if let Some(neigh) = self.neighbors.get_mut(&neighbor_addr) {
            if (delta > 0 && tp < self.transmission_powers[self.transmission_powers.len() - 1])
                || (delta < 0 && tp > self.transmission_powers[0])
            {
                neigh.control_model.1 -= delta;
            }
        }
    }

    /// Calculates the transmission power needed for a particular node/neighbor.
    fn calc_node_tp(&mut self, neighbor_addr: LoRaAddress) -> i8 {
        let neigh = self
            .neighbors
            .get(&neighbor_addr)
            .expect("calculating TP for an inexistant neighbor.");
        let tp_target = (self.lower_rssi - neigh.control_model.1) / neigh.control_model.0;
        if let Some(tp) = self
            .transmission_powers
            .iter()
            .find(|tp| (**tp as i16) >= tp_target)
        {
            return *tp;
```

```
        } else {
            return self.transmission_powers[self.transmission_powers.len() - 1];
        }
    }
}

impl ATPC for DefaultATPC {
    fn is_beacon_needed(&self) -> bool {
        return self.last_beacon.elapsed() > self.beacon_delay
            || self
                .neighbors
                .iter()
                .find(|(_, n)| n.status == NeighborStatus::Initializing)
                .is_some();
    }

    fn get_beacon_powers(&self) -> Vec<i8> {
        return self.transmission_powers.clone();
    }

    fn register_beacon(&mut self, tpi: usize, nonce: FrameNonce) {
        self.last_beacon = Instant::now();
        self.beacons.push(nonce, tpi as u8);
    }

    fn register_neighbor(&mut self, neighbor_addr: LoRaAddress) -> bool {
        // We should assure the unicity of the neighbors in the list.
        if let None = self.neighbors.get(&neighbor_addr) {
            let neigh = NeighborModel::new(neighbor_addr, self.transmission_powers.len());
            self.neighbors.push(neighbor_addr, neigh);
            true
        } else {
            false
        }
    }

    fn unregister_neighbor(&mut self, neighbor_addr: LoRaAddress) -> bool {
        return self.neighbors.pop_entry(&neighbor_addr).is_some();
    }

    fn get_tx_power(&mut self, neighbor_addr: LoRaAddress) -> i8 {
        if self.neighbors.contains(&neighbor_addr) {
            return self.calc_node_tp(neighbor_addr);
        }
        self.transmission_powers[self.default_tp as usize]
    }

    fn get_min_tx_power(&mut self, mut neighbor_addrs: Vec<LoRaAddress>) -> (i8, Vec<LoRaAddress>) {
        let mut tx_power = None;
        let mut should_update = Vec::new();
        neighbor_addrs.sort();
        for na in &neighbor_addrs {
            let tp = self.get_tx_power(*na);
            if tx_power.is_none() || tp == tx_power.unwrap() {
                should_update.push(*na);
            } else if tp > tx_power.unwrap() {
                tx_power = Some(tp);
                should_update.clear();
                should_update.push(*na);
            }
        }
        if let Some(tx_power) = tx_power {
            (tx_power, should_update)
        } else {
```

```
            (
                self.transmission_powers[self.default_tp as usize],
                neighbor_addrs,
            )
        }
    }

    fn report_successful_reception(
        &mut self,
        neighbor_addr: LoRaAddress,
        nonce: FrameNonce,
        drssi: i16,
    ) {
        if let Some(tpi) = self.beacons.get(&nonce) {
            if let Some(neigh) = self.neighbors.get_mut(&neighbor_addr) {
                neigh.rssi[*tpi as usize] = drssi;
                self.rebuid_neighbor_model(neighbor_addr);
            }
        } else {
            self.update_neighbor_model(neighbor_addr, drssi);
        }
    }

    fn report_failed_reception(&mut self, neighbor_addr: LoRaAddress) {
        self.update_neighbor_model(neighbor_addr, -30);
    }
}

/// Testing implementation.
///
/// Provides an implementation that cycles all its transmission powers across each
    transmission.
/// Moreover it does not implement beacons, and most of its operations are NO-OP.
pub struct TestingATPC {
    /// The transmission powers usable by the ATPC (and the radio).
    transmission_powers: Vec<i8>,
    /// Literally a counter of each transmission.
    counter: usize,
}

impl TestingATPC {
    /// Builds a new instance of a Testing ATPC.
    pub fn new(transmission_powers: Vec<i8>) -> Self {
        Self {
            transmission_powers,
            counter: 0,
        }
    }
}

impl ATPC for TestingATPC {
    fn is_beacon_needed(&self) -> bool {
        false
    }

    fn get_beacon_powers(&self) -> Vec<i8> {
        return vec![];
    }

    fn register_beacon(&mut self, _tpi: usize, _nonce: FrameNonce) {
        // NO-OP
    }

    fn register_neighbor(&mut self, _neighbor_addr: LoRaAddress) -> bool {
        // NO OP
        true
```

```rust
    }

    fn unregister_neighbor(&mut self, _neighbor_addr: LoRaAddress) -> bool {
        // NO OP
        true
    }

    fn get_tx_power(&mut self, _neighbor_addr: LoRaAddress) -> i8 {
        let tp = self.transmission_powers[self.counter];
        let len = self.transmission_powers.len();
        self.counter = (self.counter + 1) % len;
        return tp;
    }

    fn get_min_tx_power(&mut self, neighbor_addrs: Vec<LoRaAddress>) -> (i8, Vec<LoRaAddress>) {
        return (self.get_tx_power(*&neighbor_addrs[0]), neighbor_addrs);
    }

    fn report_successful_reception(
        &mut self,
        _neighbor_addr: LoRaAddress,
        _nonce: FrameNonce,
        _drssi: i16,
    ) {
        // NO OP
    }

    fn report_failed_reception(&mut self, _neighbor_addr: LoRaAddress) {
        // NO OP
    }
}
```

Listing 4 – radio-tipe-poc/src/device.rs

```rust
//! Definitions for the abstract device driver.
//!
//! It is the essential trait that all applications will have to use to interact with
//! the radio.
//!
//! ## Usages
//!
//! Here is a very short example of how to use [Device] to exchange messages.
//!
//! In most cases, it will run in a infinite loop to poll and push messages to the network.
//!
//! ```rust,ignore
//! pub fn spawn(&'a mut self) -> anyhow::Result<()> {
//!     // Create a Tx/Rx Client if necessary
//!     let handler = ...;
//!
//!     self.device.set_transmit_client(Box::new(handler.clone()));
//!     self.device.set_receive_client(Box::new(handler));
//!
//!     {
//!         use std::sync::mpsc::RecvTimeoutError;
//!         let mut should_transmit = false;
//!
//!         println!("Initializing ATPC (transmitting beacons)...");
//!         self.device.start_reception()?;
//!         self.device.transmit_beacon()?;
//!         self.device.start_reception()?;
//!
//!         loop {
//!             // Do something that might set should_transmit to true.
//!             // Maybe consume message from the Tx/Rx Client?
```