POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Course: Internet of Things - 5 CFU

Professor: Matteo Cesana
Students: Bruno Arena, Matteo Cavalleri
Personal IDs: 10806605, 10768765

Challenge 1

Abstract

This project focuses on the development of a commercial-grade Internet of Things (IoT) motion sensor designed for smart home integration.
The device is engineered to detect human presence through motion sensing while simultaneously monitoring ambient light levels.
By capturing these dual data points, the sensor enables a centralized Sink Node (controller) to execute intelligent lighting automation, optimizing energy efficiency and user comfort.
The resulting system provides a responsive, data-driven solution for automated residential lighting environments.

Question #1 – System requirements and Sensor node design

The idea is to implement a communication between the sensor and a central controller (Sink Node).
The sensor should transmit the motion states and the luminosity detected in a defined format:

- MOTION_DETECTED-LUMINOSITY: ZZZ
- MOTION_NOT_DETECTED-LUMINOSITY: ZZZ

After the transmission, the sensor, must enter the Deep Sleep mode for a chosen time.

We should be able to estimate the energy consumption of one cycle and estimate the total battery Life Cycle.

Hardware architecture

The sensor node consists of an ESP32 DevKit V1 board interfaced with two sensing modules:

- **PIR Motion Sensor:** provides a digital HIGH/LOW signal indicating presence detection.
- **Analog Light Sensor:** implemented in Wokwi through a slider potentiometer connected to ADC pin 34 to simulate varying light intensity.

The ESP32 also serves as the radio interface, using its integrated Wi-Fi transceiver in ESP-NOW mode. The Sink Node is assumed to be always reachable and addressed via the mandatory broadcast MAC FF:FF:FF:FF:FF:FF.

The Wokwi JSON diagram defines the exact wiring and is consistent with the sensor mapping used in the source code.

Software architecture overview

The node executes one complete sensing–transmission cycle each time it wakes from Deep Sleep. Its behavior is fully contained within the setup() function, since loop() is never executed. This design ensures a deterministic cycle structure and minimizes runtime overhead.

The execution flow is divided into five sequential phases:

1. **Sensing** (radio off)
2. **Message construction**
3. **Wi-Fi and ESP-NOW setup**
4. **ESP-NOW transmission**
5. **Entry into Deep Sleep**

Isolating these phases ensures clear functional boundaries and aligns the implementation with the specification.

Phase 1 — Motion and Ambient Light Sensing

At wake-up, the ESP32 performs environmental sensing before activating the radio subsystem. This ensures minimal energy usage during sensing, although energy aspects are formally part of Question 2.
Motion is read via the digital pin connected to the PIR sensor:

```
pinMode(PRESENCE_PIN, INPUT);
bool motion = (digitalRead(PRESENCE_PIN) == HIGH);
```

Ambient illuminance is sampled through the ADC with 12-bit resolution:

```
analogReadResolution(12);
int lux = analogRead(LIGHT_SENSOR_PIN);
```

The two raw measurements constitute the foundation of the application-level message transmitted to the Sink Node.

Phase 2 — Application Message Construction

The sensor node must assemble a single, human-readable string that encodes both motion state and the measured luminosity value. The system uses snprintf() to ensure safe formatting and compliance with the required format:

```
snprintf(message, sizeof(message), "%s-LUMINOSITY:%d",
motion ? "MOTION_DETECTED" : "MOTION_NOT_DETECTED",
lux);
```

This structure provides a lightweight and unambiguous representation of sensor data, suitable for downstream decision-making by the Sink Node.

Question 1 mandates that data must be formatted as one of the following:

- MOTION_DETECTED-LUMINOSITY:ZZZ
- MOTION_NOT_DETECTED-LUMINOSITY:ZZZ

The logic is:

1. Convert the boolean motion value into the corresponding keyword.
2. Append the light sensor value.
3. Produce a single compact string.

```
snprintf(message, sizeof(message), "%s-LUMINOSITY:%d",
motion ? "MOTION_DETECTED" : "MOTION_NOT_DETECTED",
lux);
```

This string constitutes the "application layer protocol" between the sensing node and the Sink Node.

Phase 3 — Wi Fi and ESP NOW Initialization

The radio remains OFF until communication is required. When activated, three main operations occur:

1. **Switch Wi-Fi to Station Mode**
2. **Initialize the ESP-NOW protocol stack**
3. **Register the Sink Node (broadcast MAC)**

ESP-NOW uses a connectionless model and requires no association with an access point, making it ideal for quick status updates.

```
WiFi.mode(WIFI_STA);
WiFi.setTxPower(WIFI_POWER_19_5dBm);

if (esp_now_init() != ESP_OK) {
ESP.restart();
}

esp_now_register_send_cb(OnDataSent);
```

Peer registration with broadcast MAC:

```
esp_now_peer_info_t peerInfo = {};
memcpy(peerInfo.peer_addr, sinkAddress, 6);
peerInfo.channel = 0;
```

```
peerInfo.encrypt = false;
esp_now_add_peer(&peerInfo);
```

This ensures that the message will be received by the Sink Node, which is assumed to always be active and reachable, as specified in the assignment.

Phase 4 — ESP NOW Data Transmission

Once ESP-NOW is ready, the message is transmitted as a byte array:

```
esp_now_send(sinkAddress, (uint8_t *)message, strlen(message) + 1);
```

The system waits for the send callback to confirm completion or until a timeout occurs. This ensures correct delivery semantics in the simulation environment and matches typical behavior in embedded sensor networks.

```
void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
tTxEnd = micros();
sendDone = true;
}
```

A timeout prevents the system from waiting indefinitely:

```
unsigned long startWait = millis();
while (!sendDone && (millis() - startWait < 1000)) {
yield();
}
```

This ensures robust behavior even if ACKs are delayed or lost.

Phase 5 — Transition to Deep Sleep

After transmitting the required message, the node's cycle is complete. Per Requirement 1, the node must enter Deep Sleep immediately. The logic is:

1. Turn Wi-Fi OFF
2. Configure a timer wake-up
3. Enter Deep Sleep state

Once the message has been delivered, the Wi-Fi subsystem is deactivated and the ESP32 enters Deep Sleep.

Code reference:

```
WiFi.mode(WIFI_OFF);
esp_sleep_enable_timer_wakeup(SLEEP_TIME_US);
esp_deep_sleep_start();
```

The mechanism behind sleep mode is essential to Question 1: after each transmission, the node must transition into a low-power suspended state, completing the operational cycle.

The exact value of the deep sleep time is determined from the last two digits of the team leader's personal code. The assignment specifies the following formula:

$$X = \frac{(AB \ \% \ 50) + 5}{10}$$

where **AB** represents the last two digits. In this case, we have considered the following personal code: 10806605.

This value determines the time, expressed in seconds, during which the node remains in Deep Sleep before restarting a new sensing–transmission cycle. The calculated duration is then converted into microseconds and passed to the ESP32 deep-sleep timer. This parameter completes the duty-cycle definition required for the sensor node's operation and ensures that each cycle maintains a fixed and reproducible period.

Question #2 - Introduction

The second part of the challenge focuses on the quantitative characterization of the sensor node's longevity. While the first requirement ensures that the node performs all sensing and communication tasks correctly, Requirement 2 aims to translate these operations into measurable power and energy figures.
The objective is to determine how much energy each phase of the cycle consumes—sensing, message construction, Wi-Fi/ESP-NOW setup, transmission, idle time, and Deep Sleep—and ultimately estimate the expected lifetime of the device when powered by a battery of predefined energy capacity.
To achieve this, the challenge mandates a two-step methodology. First, the temporal duration of each operational phase must be obtained directly from the implemented code running in Wokwi. These time intervals provide a detailed breakdown of how long the ESP32 spends in each state, such as radio-on versus radio-off, CPU-active versus CPU-idle, and Deep Sleep. Second, these measured durations must be combined with the average power values provided through dedicated CSV datasets. Each dataset corresponds to a specific hardware state of the ESP32 (e.g., transmission at different power levels, idle mode, Wi-Fi enabled, Deep Sleep), allowing the construction of a complete power-versus-time model of the system.

The purpose of Requirement 2 is therefore twofold:

1. **Derive the energy consumed by the node for one full sensing–transmission–sleep cycle**, using the physical relation

$$E = P * \Delta t$$

2. **Estimate the operational lifetime of the device**, based on a virtual battery whose energy content is defined by the last digits of the team leader's personal code (in this case: 10806605).

This requirement shifts the focus from pure functionality to performance analysis. By quantifying how the node behaves energetically, it becomes possible to evaluate its efficiency, identify potential bottlenecks, and later justify architectural improvements aimed at reducing consumption. The outcome of this analysis provides the foundation for the reasoning developed in the subsequent challenge sections.

The energy performances of the ESP32 are described in the CSV files provided for the challenge.
There are three different CSVs:

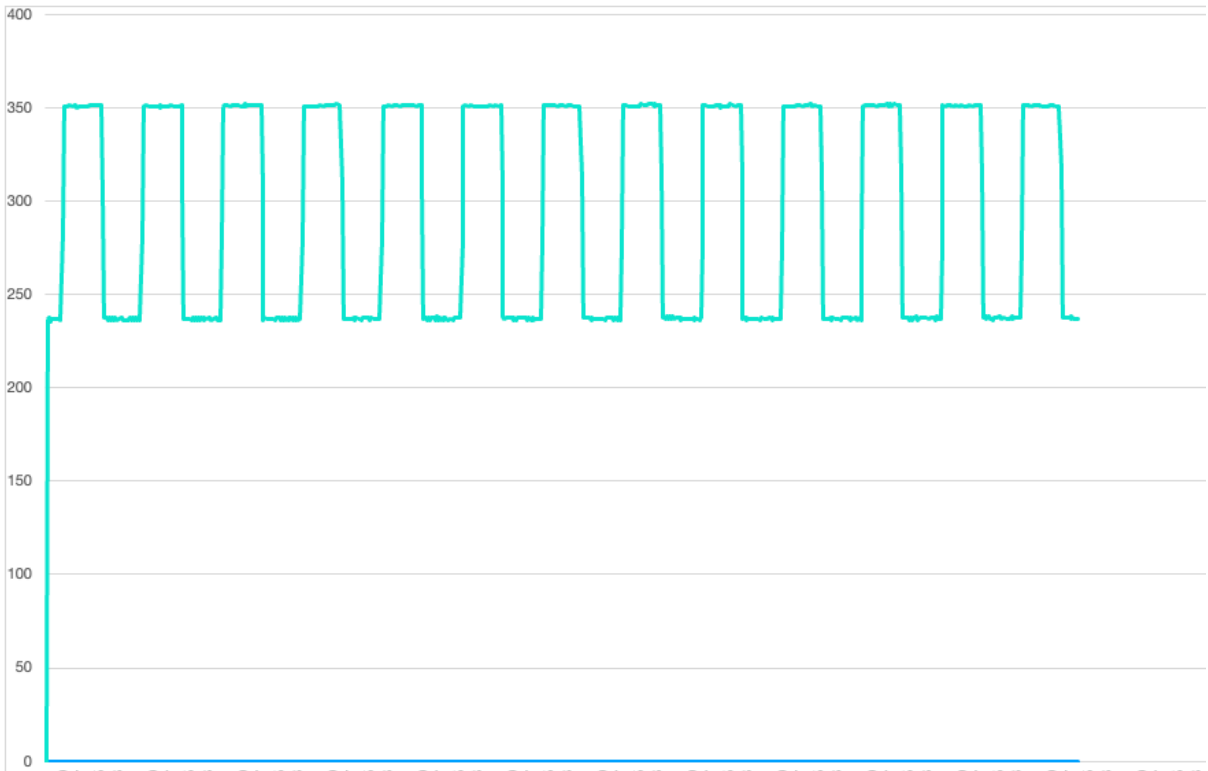
- Sensor - Read
- Sender
- Deep Sleep

These files describe the energy profiles of all transmission states.

Sensor – Read

In this file is described the Sensing/Building phases.
The Sensing Phase is the very first step of the cycle and represents the "physical" interaction with the world. During this stage, the ESP32 communicates with the hardware components—specifically the PIR motion sensor and the photoresistor.

The Building Phase a purely logical and computational step where the CPU takes those "raw" variables and transforms them into a structured message.

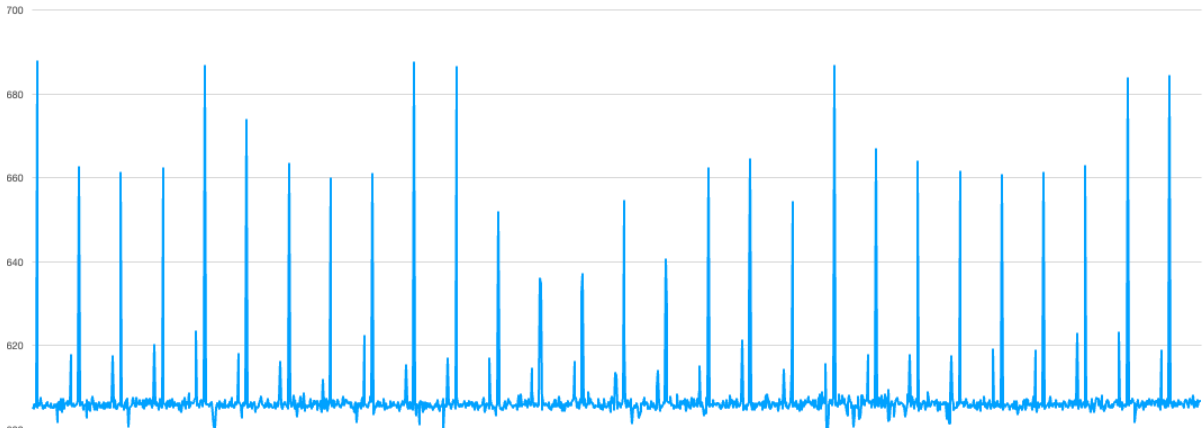


Wi-Fi ON

Once the data is ready, the system enters the Wi-Fi Setup Phase. This is the moment the "energy cost" increases, as the internal radio hardware is powered on.

Sender

The Transmission Phase is the core objective of the active cycle. Here, the "built" message is pushed through the ESP-NOW layer.
The power factor is setup at 19,5 dBm.



Wi-Fi OFF

The processor then stays in a brief IDLE state, printing the output time and then is ready to enter the deep sleep to save energy.

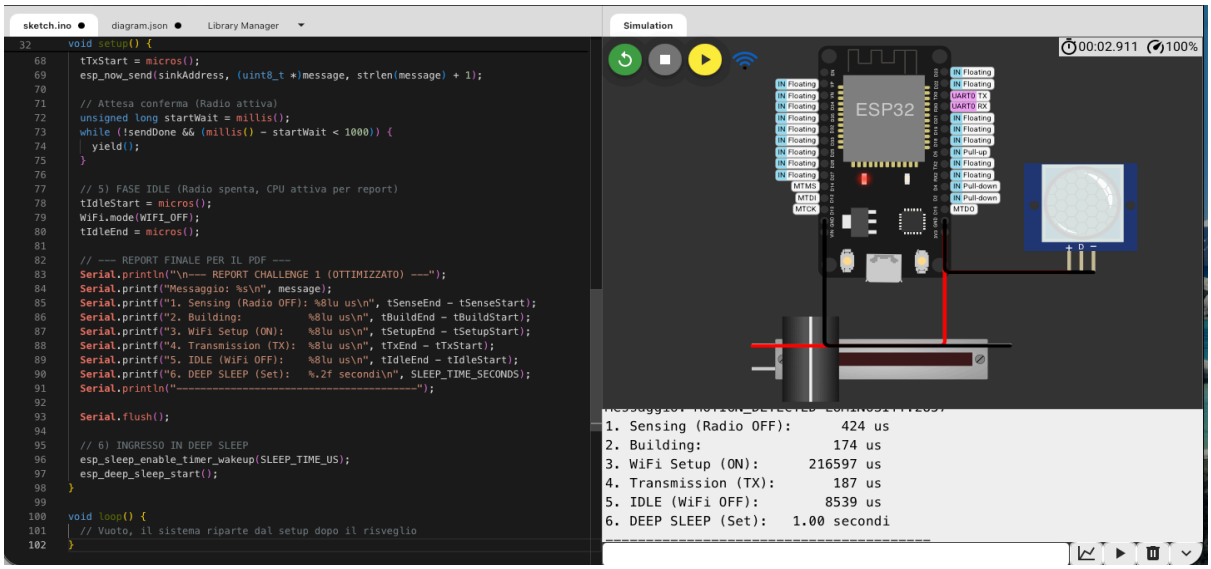
Deep Sleep

The final stage is the Deep Sleep Phase, the ultimate tool for battery conservation. The ESP32 clears its volatile memory and shuts down almost every internal component, including the dual cores and most of the RAM.

Question #2 – Cycle time

Analysing how the code works is possible to estimate a working cycle for the ESP32.

Analysing different cycle times is possible to conclude that this are the mean times:



Sensing	$474\ \mu s$
Building	$174\ \mu s$
Wi-Fi ON (Setup)	$216.597\ \mu s$
Transmission	$187\ \mu s$
Wi-Fi OFF (IDLE)	$8.539\ \mu s$
Deep Sleep	$1\ s$
Total Cycle Time	$\mathbf{1,23\ s}$

Question #2 – Energy consumption

By studying the CSVs provided we identify the mean values of power consumption of the different phases:

Sensor – Read	$350{,}86\ mW$
Wi-Fi On	$633{,}00\ mW$
Transmission	$687{,}93\ mW$
Wi-Fi OFF	$249{,}54\ mW$
Deep Sleep	$45{,}64\ mW$

By integrating the power consumption with the cycle times, we obtained the total energy consumption per cycle:

$$E = P * \Delta t$$

Sensor – Read	$0{,}21\ mJ$
Wi-Fi On	$137{,}11\ mJ$
Transmission	$0{,}13\ mJ$
Wi-Fi OFF	$2{,}13\ mJ$
Deep Sleep	$45{,}64\ mJ$
Total Energy Consumption	$\mathbf{0{,}19\ J}$

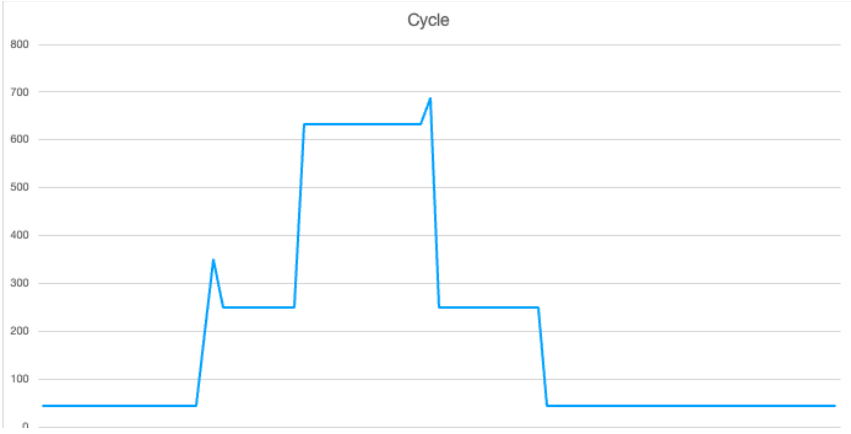


Figure 1 - Cycle

Question #2 – Lifetime estimation

As requested, the battery energy is computed as follow, using the personal IDs last four digits:

$$Y = (6605\ \%\ 5.000) + 15.000 = 16.605\ J$$

The total lifetime of the battery can be computed calculating the cycle total time and how many cycle the battery can support:

$$T_{life} = \frac{16.605\ J}{0{,}19\ J} * \frac{1{,}23\ s}{3.600\ \frac{s}{h}} = 30{,}53\ h$$

Results

The operational autonomy of the node was estimated by integrating the current consumption across all functional phases: Sensing, Transmission (at 19.5 dBm), and Deep Sleep.

- ***Duty Cycle Analysis***: The high-power consumption phase represents only a fraction of a second, while majority of the consumption is on the Wi-Fi setup.
- ***Estimated Service Life***: Based on a standard battery, the calculated service life is 30,53 hours.

The most significant factor limiting autonomy is the frequency of the transmission cycle. According to the firmware logic, the sleep duration is calculated using a specific person code (AB=05), which results in a 1-second sleep interval.
By waking up every 1,23 seconds creates a massive energy overhead because of:

- Energy drain during the Wi-Fi initialization phase;
- Waking up so frequently costs a lot in terms of energy by setting up everything again.

Question #3 - Introduction

The third part of the challenge focuses on the analysis and enhancement of the sensor node's efficiency. After defining the operating cycle and estimating the energy consumption of each phase, this section examines how the system can be optimized to reduce its overall power demand while preserving its functional behaviour. The goal is to identify which stages of the cycle contribute most to energy usage and determine how modifications at the software, timing, or configuration level can lead to measurable improvements.
The analysis is based on the timing measurements and power profiles obtained previously. By combining these elements, it becomes possible to pinpoint inefficiencies such as prolonged radio-on intervals, unnecessary CPU activity, or non-optimal duty-cycling strategies. The improvements proposed in this part are therefore grounded in empirical data and focus on enhancing the node's operational sustainability.
This section ultimately aims to refine the original implementation, assess the impact of the proposed modifications, and compare the optimized version with the baseline. In doing so, it establishes a clear link between design choices, system behaviour, and energy performance, providing a structured foundation for targeted optimization.

Analysis of the baseline system

This section provides an overview of the energy profile of the original implementation. By combining the measured execution times of each operational phase with the corresponding power levels provided in the reference datasets, it becomes possible to quantify the energy cost of a full sensing–transmission cycle. The aim of this chapter is to highlight which system components and execution phases contribute most significantly to total consumption, establishing a clear baseline for subsequent optimization.

In the baseline implementation, the overall energy profile is primarily determined by how long the ESP32 remains in its high-power radio states. While the sensing and message-building phases execute very quickly and with the radio turned off, the Wi-Fi and ESP-NOW operations dominate both the execution time and the energy consumption of each cycle.
In the original version of the firmware (Question 1), the radio is initialized at maximum transmission power:

```
WiFi.mode(WIFI_STA);
WiFi.setTxPower(WIFI_POWER_19_5dBm);
esp_now_init();
```

esp_now_add_peer(&peerInfo);

This configuration results in two major contributors to energy usage:

1. **Wi-Fi/ESP-NOW setup:** turning on the Wi-Fi module and initializing ESP-NOW requires the ESP32 to activate the entire RF subsystem, stabilize internal clocking, and configure the protocol stack. This phase consistently appears as one of the longest in the measured timing data. Its duration, combined with the high-power draw of the Wi-Fi hardware, makes it the largest single contributor to total energy consumption.
2. **Transmission at maximum output power:** the actual packet transmission, although short in duration, occurs at the highest transmit power allowed by the ESP32 (19.5 dBm). Even with a brief TX window, the instantaneous consumption is significant, increasing the overall energy per cycle.

The other phases—sensing, building, and the short idle period—have a minimal impact on energy usage. The sensing portion consists only of an ADC read and a digital input read, operations that complete in a few hundred microseconds with negligible current draw. Message construction through snprintf() likewise represents an extremely short CPU-only phase. The idle phase, which occurs after disabling Wi-Fi using:

WiFi.mode(WIFI_OFF);

adds only a small amount of overhead, as the CPU remains active briefly before entering Deep Sleep.

Overall, the baseline profile is characterized by a heavy reliance on high-power radio operation, with the Wi-Fi setup and transmission stages being responsible for most of the energy consumed during each cycle. This observation forms the basis for the optimization strategies introduced in the subsequent sections, where efforts are focused on reducing radio usage, shortening active intervals, and eliminating unnecessary transmissions.

Optimization opportunities & Proposed improvements

Based on the baseline analysis, several aspects of the original implementation present clear opportunities for energy optimization. The goal is to reduce the time spent in high-power states and eliminate unnecessary transmissions, while preserving the functional behaviour of the sensor node. By comparing the

baseline logic with the optimized version, three main areas of intervention emerge as the most impactful: **radio activity reduction**, **transmission power adjustment**, and **event-driven duty-cycling enhancements**.

The first inefficiency identified in the baseline system concerns the unconditional activation of the Wi-Fi subsystem on every cycle. In the original implementation, the ESP32 enables the radio and initializes the ESP-NOW protocol regardless of whether the sensed data has changed:

```
WiFi.mode(WIFI_STA);
WiFi.setTxPower(WIFI_POWER_19_5dBm);
esp_now_init();
```

This behaviour implies that even when no meaningful environmental variation is detected, the radio is still powered on, initialized, and used for transmitting redundant messages. Since Wi-Fi initialization is one of the most expensive phases in terms of time and power, eliminating unnecessary activations represents a major optimization opportunity.

A second area of improvement arises from the use of **maximum transmit power (19.5 dBm)**. While this ensures robust communication, it is excessive for a short-range scenario such as a Wokwi simulation or a typical smart-home environment. The optimized code reduces this value:

```
WiFi.setTxPower(WIFI_POWER_2dBm);
```

This change significantly decreases the current consumption during transmission while maintaining reliable communication in the target context.

The third major optimization concerns the introduction of an **event-driven transmission strategy**. In the baseline implementation, a message is transmitted at every wake-up interval, even when the measured light level or motion state has not changed. The optimized version introduces threshold-based and state-based checks:

```
bool luxChanged = (abs(lux - lastLux) > LUX_THRESHOLD);
bool motionChanged = (motion != lastMotion);
```

If no meaningful variation is detected, the system returns immediately to Deep Sleep without enabling Wi-Fi:

```
if (!motion && !luxChanged && lastLux != -1) {
entraInDeepSleep();
return;
}
```

This modification prevents redundant transmissions, directly reducing the number of high-power events in the lifecycle of the node.

Lastly, the optimized version introduces **external wake-up triggers** through the PIR sensor:

```
esp_sleep_enable_ext0_wakeup((gpio_num_t)PRESENCE_PIN, 1);
```

This modification enables the ESP32 to wake immediately when motion is detected, reducing the need for frequent periodic wake-ups solely for polling. This enhances responsiveness while simultaneously lowering average energy usage.
Taken together, these improvements target the main inefficiencies identified in the baseline code: unnecessary radio usage, excessive transmit power, and strictly periodic sampling without event awareness. The next section details how these opportunities have been translated into practical modifications within the optimized implementation.

Implementation of the optimized version

The optimized implementation introduces several targeted modifications to reduce the overall energy consumption of the sensing node while preserving its functional behavior. These improvements translate into concrete changes in the firmware structure, event-handling logic, and radio configuration. Compared to the baseline implementation, the optimized system significantly reduces the number of radio activations, shortens time spent in high-power states, and limits unnecessary transmissions.

A first major change is the introduction of **state-based and threshold-based transmission logic**. In the original version, a message was transmitted at every

wake-up cycle, regardless of whether the environment had changed. The optimized version uses two persistent RTC variables (lastLux and lastMotion) combined with a luminosity threshold to evaluate whether the new readings represent a significant variation. In code:

```
bool luxChanged = (abs(lux - lastLux) > LUX_THRESHOLD);
bool motionChanged = (motion != lastMotion);
```

If neither condition is true, the device avoids enabling the Wi-Fi module entirely and returns directly to Deep Sleep:

```
if (!motion && !luxChanged && lastLux != -1) {
entraInDeepSleep();
return;
}
```

This modification eliminates redundant transmissions and prevents the radio from being activated unnecessarily, which is one of the most effective energy-saving strategies applied.

A second important adjustment concerns the **transmission power level**. In the baseline implementation, the ESP32 transmitted at maximum output power (19.5 dBm), which is excessive for short-range communication. The optimized code lowers this value to 2 dBm:

```
WiFi.setTxPower(WIFI_POWER_2dBm);
```

This single change reduces the instantaneous power draw during both the setup and transmission phases while maintaining reliable communication in the expected operating scenario.

Another key improvement involves the addition of **external wake-up support** through the PIR sensor. While the baseline system woke up exclusively using a timer, the optimized version configures an external wake-up trigger:

```
esp_sleep_enable_ext0_wakeup((gpio_num_t)PRESENCE_PIN, 1);
```

With this enhancement, the node can remain asleep for longer periods when the environment is stable and wake up immediately only when motion occurs. This

enables a more responsive yet energy-efficient behavior, as the system no longer relies solely on periodic polling to detect motion events.

The optimized version also reorganizes the execution flow to reduce overhead. For example, the deep-sleep transition is encapsulated in a dedicated function (entraInDeepSleep()), improving code clarity and ensuring consistent configuration of wake-up sources. Additionally, Wi-Fi initialization is now performed only after verifying that a transmission is necessary, preventing the system from performing expensive setup operations when they are not required.

Finally, the structure of the sensing and reporting phases remains similar, but their interaction with the new transmission logic is streamlined. The message is still constructed in the same format, but only after confirming that an update must be sent:

```
snprintf(message, sizeof(message), "%s-LUMINOSITY:%d",
motion ? "MOTION_DETECTED" : "MOTION_NOT_DETECTED", lux);
```

In summary, the optimized implementation builds upon the baseline logic but redefines the key decision points that determine when the radio is activated and how the ESP32 enters and exits sleep. These modifications produce a more adaptive and energy-aware system, reducing the number of high-power events and improving the node's long-term efficiency without compromising its operational objectives.

Results & Design consideration

The optimized implementation results in a clear improvement in the overall energy efficiency of the sensor node. By avoiding unnecessary radio activations, reducing transmission power, and enabling event-driven wake-ups, the system minimizes the time spent in high-power states while preserving its functional behavior. The introduction of conditional transmission logic significantly decreases the number of full sensing–setup–transmission cycles in stable conditions, and the reduction of Wi-Fi transmit power lowers the energy required for each individual message. The external wake-up mechanism further extends the time spent in Deep Sleep, reducing idle wake-ups and enhancing responsiveness to motion events.

Taken together, these modifications shift the node from a strictly periodic and energy-expensive workflow to a more adaptive and efficient design. The optimized version achieves a meaningful reduction in energy consumption without compromising reliability or sensing performance, demonstrating how thoughtful architectural adjustments can substantially improve the long-term sustainability of an IoT device.

Full code reference

In the following chapter, the complete firmware listings used in Question 1 and Question 3 are reported in full. These code references consolidate all the implementation details discussed throughout the report and serve as the definitive source for the baseline and optimized sensor-node logic.

Question 1 – Original Code

```
#include <WiFi.h>
#include <esp_now.h>
#include <esp_sleep.h>
#include <string.h>
#include <stdio.h>

int LIGHT_SENSOR_PIN = 34;
int PRESENCE_PIN     = 27;

// --- CALCOLO DEEP SLEEP  ---
int AB = 05;
float SLEEP_TIME_SECONDS = ((AB % 50) + 5) / 10.0;
uint64_t SLEEP_TIME_US = (SLEEP_TIME_SECONDS * 1000000ULL);
```

```
// Indirizzo di Broadcast
uint8_t sinkAddress[] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
char message[64];

unsigned long tSetupStart, tSetupEnd, tSenseStart, tSenseEnd, tBuildStart, tBuildEnd;
unsigned long tTxStart, tTxEnd, tIdleStart, tIdleEnd;
bool sendDone = false;

// Callback per conferma invio
void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
  tTxEnd = micros(); // Fine fase TX al momento dell'ACK
  sendDone = true;
}

void setup() {
  Serial.begin(115200);

  // 1) FASE SENSING
  tSenseStart = micros();
  pinMode(PRESENCE_PIN, INPUT);
  analogReadResolution(12);
```

```
  int lux = analogRead(LIGHT_SENSOR_PIN);
  bool motion = (digitalRead(PRESENCE_PIN) == HIGH);
  tSenseEnd = micros();

  // 2) FASE BUILDING
  tBuildStart = micros();
  snprintf(message, sizeof(message), "%s-LUMINOSITY:%d",
        motion ? "MOTION_DETECTED" : "MOTION_NOT_DETECTED",
lux);
  tBuildEnd = micros();

  // 3) FASE SETUP WIFI & ESP-NOW (Accensione Radio)
  tSetupStart = micros();
  WiFi.mode(WIFI_STA);
  WiFi.setTxPower(WIFI_POWER_19_5dBm);

  if (esp_now_init() != ESP_OK) {
    ESP.restart();
  }

  esp_now_register_send_cb((esp_now_send_cb_t)OnDataSent);
```

```
esp_now_peer_info_t peerInfo = {};
memcpy(peerInfo.peer_addr, sinkAddress, 6);
peerInfo.channel = 0;
peerInfo.encrypt = false;
esp_now_add_peer(&peerInfo);
tSetupEnd = micros();

// 4) FASE TRANSMISSION
tTxStart = micros();
esp_now_send(sinkAddress, (uint8_t *)message, strlen(message) + 1);

// Attesa conferma (Radio attiva)
unsigned long startWait = millis();
while (!sendDone && (millis() - startWait < 1000)) {
  yield();
}

// 5) FASE IDLE
tIdleStart = micros();
WiFi.mode(WIFI_OFF);
tIdleEnd = micros();
```

```
  // --- REPORT FINALE PER IL PDF ---
  Serial.println("\n--- REPORT CHALLENGE 1 (OTTIMIZZATO) ---");
  Serial.printf("Messaggio: %s\n", message);
  Serial.printf("1. Sensing (Radio OFF): %8lu us\n", tSenseEnd - tSenseStart);
  Serial.printf("2. Building:           %8lu us\n", tBuildEnd - tBuildStart);
  Serial.printf("3. WiFi Setup (ON):    %8lu us\n", tSetupEnd - tSetupStart);
  Serial.printf("4. Transmission (TX):  %8lu us\n", tTxEnd - tTxStart);
  Serial.printf("5. IDLE (WiFi OFF):    %8lu us\n", tIdleEnd - tIdleStart);
  Serial.printf("6. DEEP SLEEP (Set):          %.2f secondi\n",
SLEEP_TIME_SECONDS);
  Serial.println("---------------------------------------");

  Serial.flush();

  // 6) INGRESSO IN DEEP SLEEP
  esp_sleep_enable_timer_wakeup(SLEEP_TIME_US);
  esp_deep_sleep_start();
}

void loop() {
 }
```

Question 3 – Upgraded Code

```
#include <WiFi.h>
#include <esp_now.h>
#include <esp_sleep.h>
#include <string.h>
#include <stdio.h>

int LIGHT_SENSOR_PIN = 34;
int PRESENCE_PIN     = 27;

// Soglia di variazione luminosità
const int LUX_THRESHOLD = 100;

// --- VARIABILI RTC (Mantengono il valore durante il Deep Sleep) ---
RTC_DATA_ATTR int lastLux = -1;
RTC_DATA_ATTR bool lastMotion = false;

// --- CALCOLO DEEP SLEEP ---
int AB = 05;
float SLEEP_TIME_SECONDS = (float)((AB % 50) + 5) / 10.0;
```

```
uint64_t SLEEP_TIME_US = (uint64_t)(SLEEP_TIME_SECONDS * 1000000ULL);


// Configurazione ESP-NOW
uint8_t sinkAddress[] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
char message[64];


unsigned long tSetupStart, tSetupEnd, tSenseStart, tSenseEnd, tBuildStart, tBuildEnd;
unsigned long tTxStart, tTxEnd, tIdleStart, tIdleEnd;
volatile bool sendDone = false;


// Callback per conferma invio
void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
  tTxEnd = micros();
  sendDone = true;
}


void setup() {
  Serial.begin(115200);


  // 1) FASE SENSING (Radio OFF per risparmio energetico)
```

```
  tSenseStart = micros();
  pinMode(PRESENCE_PIN, INPUT);
  analogReadResolution(12);
  int lux = analogRead(LIGHT_SENSOR_PIN);
  bool motion = (digitalRead(PRESENCE_PIN) == HIGH);
  tSenseEnd = micros();

  // --- LOGICA DI RISPARMIO AVANZATA ---
  bool luxChanged = (abs(lux - lastLux) > LUX_THRESHOLD);
  bool motionChanged = (motion != lastMotion);

  // Se non ci sono variazioni rilevanti, non accendiamo neanche la radio
  if (!motion && !luxChanged && lastLux != -1) {
    Serial.println("\n[INFO] Nessuna variazione rilevante. Torno in Deep Sleep immediato.");
    entraInDeepSleep();
    return;
  }

  // Aggiorniamo i valori RTC
  lastLux = lux;
```

```
  lastMotion = motion;


  // 2) FASE BUILDING
  tBuildStart = micros();
  snprintf(message, sizeof(message), "%s-LUMINOSITY:%d",
          motion  ?  "MOTION_DETECTED"  :  "MOTION_NOT_DETECTED",
lux);
  tBuildEnd = micros();


  // 3) FASE SETUP WIFI & ESP-NOW (Efficientamento: Potenza ridotta)
  tSetupStart = micros();
  WiFi.mode(WIFI_STA);
  WiFi.setTxPower(WIFI_POWER_2dBm);


  if (esp_now_init() != ESP_OK) {
    ESP.restart();
  }


  esp_now_register_send_cb((esp_now_send_cb_t)OnDataSent);
  esp_now_peer_info_t peerInfo = {};
  memcpy(peerInfo.peer_addr, sinkAddress, 6);
  peerInfo.channel = 0;
```

```
peerInfo.encrypt = false;

esp_now_add_peer(&peerInfo);

tSetupEnd = micros();


// 4) FASE TRANSMISSION

tTxStart = micros();

esp_now_send(sinkAddress, (uint8_t *)message, strlen(message) + 1);


// Attesa conferma ACK

unsigned long startWait = millis();

while (!sendDone && (millis() - startWait < 1000)) {

  yield();

}


// 5) FASE IDLE

tIdleStart = micros();

WiFi.mode(WIFI_OFF);

tIdleEnd = micros();


// --- REPORT FINALE  ---

Serial.println("\n--- REPORT CHALLENGE 1 (OTTIMIZZATO) ---");
```

```
  Serial.printf("Messaggio: %s\n", message);
  Serial.printf("1. Sensing (Radio OFF): %8lu us\n", tSenseEnd - tSenseStart);
  Serial.printf("2. Building:           %8lu us\n", tBuildEnd - tBuildStart);
  Serial.printf("3. WiFi Setup (ON):    %8lu us\n", tSetupEnd - tSetupStart);
  Serial.printf("4. Transmission (TX):  %8lu us\n", tTxEnd - tTxStart);
  Serial.printf("5. IDLE (WiFi OFF):    %8lu us\n", tIdleEnd - tIdleStart);
  Serial.printf("6. DEEP SLEEP (Set):   %.2f secondi\n", SLEEP_TIME_SECONDS);
  Serial.println("---------------------------------------");

  Serial.flush();

  entraInDeepSleep();
}

void entraInDeepSleep() {
  // Efficientamento: Risveglio istantaneo se il sensore PIR rileva movimento (GPIO 27)
  esp_sleep_enable_ext0_wakeup((gpio_num_t)PRESENCE_PIN, 1);
  // Risveglio a tempo (Timer)
  esp_sleep_enable_timer_wakeup(SLEEP_TIME_US);
  esp_deep_sleep_start();
```

```
}


void loop() {

}
```