AGONSANOU Lumière
AGOSSA Kewen

RAPPORT PROJET CONCEPTION ET RÉALISATION D'UN OBJET CONNECTÉ

Réalisé par : Lumière AGONSANOU

Kewen AGOSSA

PROFESSEUR : BOUVIER Thierry & PERDRIAU Richard

ESEO 2025

SOMMAIRE

INTRODUCTION

Partant du constat qu'aucun projet collaboratif n'était proposé à l'ESEO, Kewen et moi avons décidé de travailler sur un sous-ensemble du projet d'optimisation du vélo mobile.

Pour cela, nous avons choisi d'associer plusieurs capteurs et actionneurs afin de concevoir un système intelligent et embarqué.

Description du système

- **Protocole de communication :** Bluetooth Low Energy (BLE)
- **Capteurs (en I2C) :** BMP180 (pression), MAX30102 (fréquence cardiaque), MLX90614 (température infrarouge)
- **Affichage :** Écran OLED SSD1309 en SPI
- **Plateforme de développement :** nRF52832 (PCA10040) avec le Nordic SDK

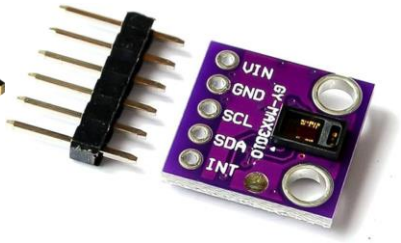
Fonctionnement général

Le système repose sur une **carte nRF centrale** qui émet un signal BLE pour se connecter à une **carte nRF périphérique**. Une fois la connexion établie, la centrale envoie périodiquement les données collectées (depuis les capteurs) vers la carte périphérique. Ces données sont ensuite affichées en temps réel sur l'écran OLED SSD1309.

AGONSANOU Lumière
AGOSSA Kewen

Présentation et descriptions des composants

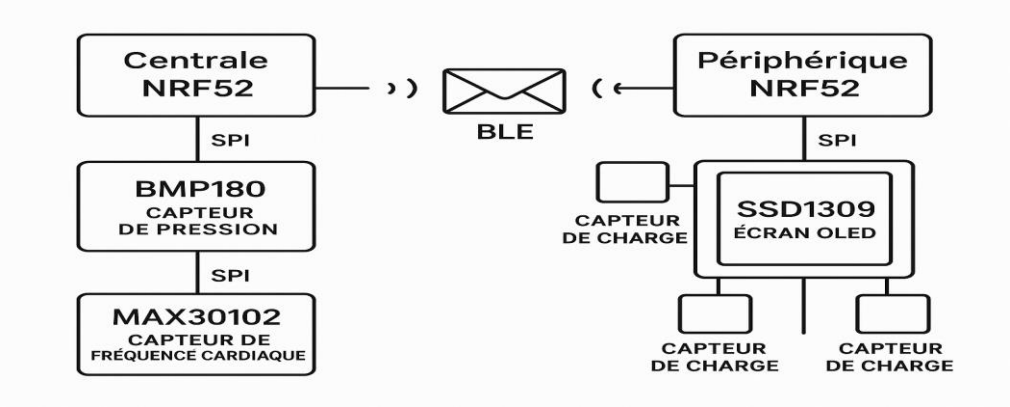
NRF52



MAX30102

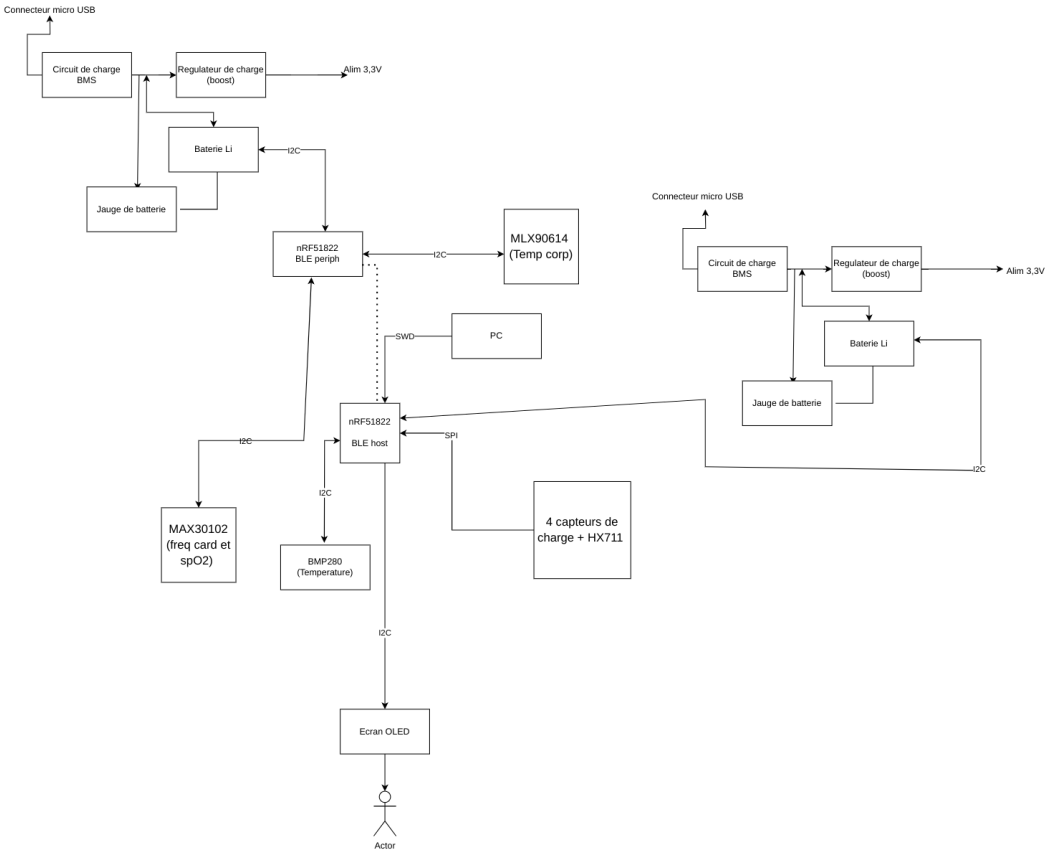
BMP180

MLX90614



AGONSANOU Lumière
AGOSSA Kewen

Schéma synoptique



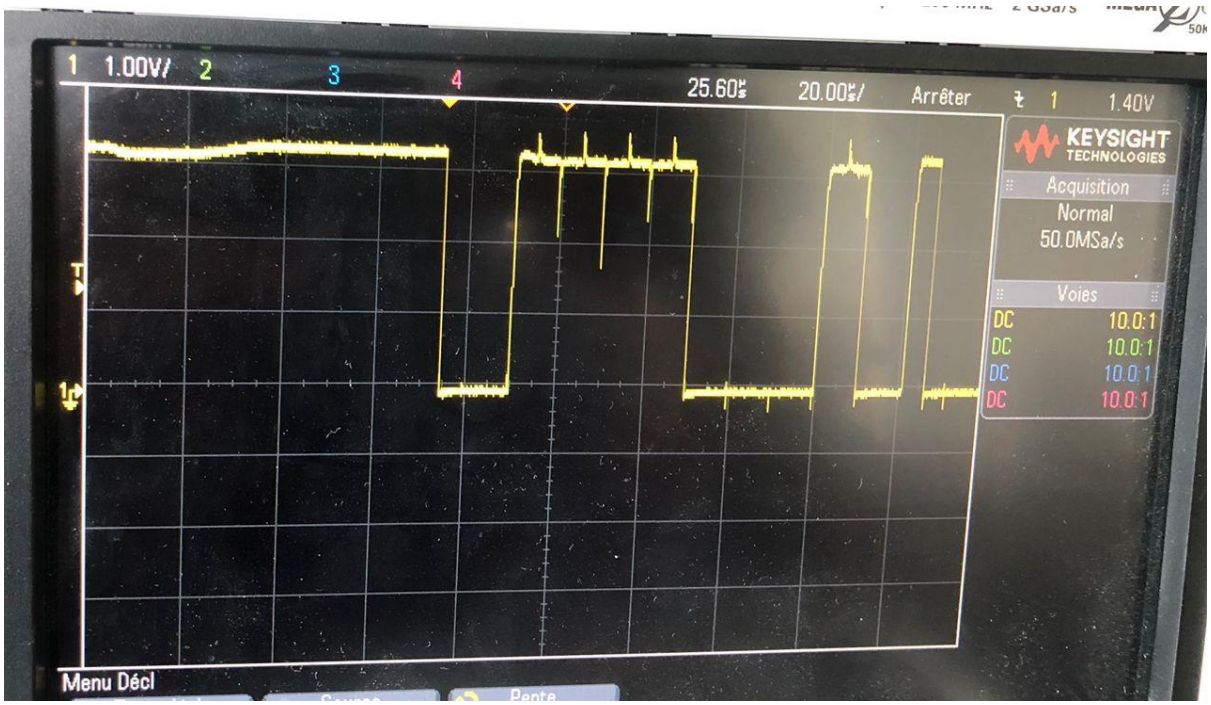
INCREMENT 1: TEST DE LA BMP180

Aperçu du capteur

Le BMP180 est un capteur barométrique digital de Bosch permettant de mesurer la **pression atmosphérique**, la **température** et d'estimer l'**altitude**. Il communique via le bus **I2C** (adresse fixe 0x77).

DEVELOPPEMENT DU CAPTEUR :

Tout d'abord nous avons essayé de capturer une tram I2C adressé au 0X77



En effet, nous sommes désormais capables d'envoyer des requêtes I2C de lire et décrire dans plusieurs registres pour récupérer les données du capteur.

ARCHITECTURE :

Le BMP180 fonctionne comme tout autre capteur I2C : il suffit d'effectuer des lectures/écritures sur des registres bien définis. Bien que Bosch fournisse une librairie complète, **il est important pour un ingénieur de comprendre le fonctionnement interne du capteur**, sans dépendre uniquement du code fourni.

Fonctionnement par étapes

1. Lecture des données d'étalonnage

- Stockées dans des registres de 0xAA à 0xBF (22 octets)
- Ce sont des coefficients (AC1 à AC6, B1, B2, MB, MC, MD)
- Nécessaires pour calculer la température et la pression corrigées

2. Mesure de température (UT)

- Écrire 0x2E dans le registre 0xF4 pour lancer une mesure

- Attendre 5 ms
- Lire 2 octets (registres 0xF6 et 0xF7) pour obtenir UT

3. Mesure de pression (UP)

- Écrire 0x34 + (OSS << 6) dans 0xF4 (OSS = précision de 0 à 3)
- Attendre 8 ms (OSS = 0) jusqu'à 26 ms (OSS = 3)
- Lire 3 octets (0xF6, 0xF7, 0xF8) pour obtenir UP

4. Formules de calcul (d'après datasheet Bosch)

a. Température

```
X1 = (UT - AC6) * AC5 >> 15;
X2 = (MC << 11) / (X1 + MD);
B5 = X1 + X2;
T = (B5 + 8) >> 4 ; // T en 0.1 °C
```

b. Pression

```
B6 = B5 - 4000;
X1 = (B2 * (B6 * B6 >> 12)) >> 11;
X2 = (AC2 * B6) >> 11;
X3 = X1 + X2;
B3 = ((AC1 * 4 + X3) << OSS + 2) >> 2;

X1 = (AC3 * B6) >> 13;
X2 = (B1 * (B6 * B6 >> 12)) >> 16;
X3 = ((X1 + X2) + 2) >> 2;
B4 = AC4 * (uint32_t)(X3 + 32768) >> 15;
B7 = ((uint32_t)UP - B3) * (50000 >> OSS);

if (B7 < 0x80000000)
   p = (B7 << 1) / B4;
else
   p = (B7 / B4) << 1;
```

```
X1 = (p >> 8) * (p >> 8);
X1 = (X1 * 3038) >> 16;
X2 = (-7357 * p) >> 16;
p = p + ((X1 + X2 + 3791) >> 4); // en Pa
```

5. Altitude (optionnelle)

```
Altitude = 44330 * (1 - pow (pressure / 101325.0, 0.1903));
```

6. Timing et délais

- Température : 5 ms
- Pression (OSS=0) : 8 ms
- Pression (OSS=3) : 26 ms

Conclusion

Le BMP180 est un capteur fiable et précis, mais qui nécessite un **traitement logiciel complexe**. Pour une intégration efficace, il est crucial de **maîtriser toutes les étapes du protocole**, depuis la lecture des registres jusqu'au calcul final.

Nous sommes ainsi capables de lire les données par les logs nRF :

```
<info> app: BMP180: T = 26.0 C | P = 1012 hPa
```

INCREMENT 2 : TEST DE LA MAX30102

1. Aperçu du capteur

Le **MAX30102** est un capteur optique intégré permettant de mesurer la **fréquence cardiaque** et la **saturation en oxygène du sang (SpO_2)**. Il est basé sur la photo pléthysmographie (PPG) et utilise deux LEDs (rouge et infrarouge) pour analyser le flux sanguin. Il communique via le bus **I2C**, avec une adresse par défaut de **0x57**.

2. Développement et communication I2C

Nous avons d'abord vérifié la communication I2C en détectant une trame envoyée à **0x57**.

Nous avons ensuite initialisé le capteur et accédé à ses registres internes pour configurer les LED, le mode opératoire (SpO_2, HR ou Multi-LED), ainsi que la fréquence d'échantillonnage.

3. Architecture de fonctionnement

Le MAX30102 intègre :

- Un convertisseur analogique-numérique (ADC) 18 bits
- Deux LEDs (rouge et IR)
- Une photodiode
- Un système de filtrage et de compensation des interférences

Le capteur est conçu pour capturer les données optiques dans différents contextes (au repos, en mouvement) avec une grande sensibilité. La configuration correcte des **timings**, **LEDs**, et **slots** est donc cruciale.

4. Fonctionnement par étapes

a. Configuration initiale

- Soft reset via le registre **MODE_CONFIG (0x09)** → bit 6 à 1
- Attendre la fin du reset (bit 6 devient 0)

b. Configuration des LEDs

- LED rouge : registre **LED1_PA (0x0C)**, valeur typique : 0x24 (courant ~7 mA)

- LED IR : registre **LED2_PA (0x0D)**, valeur similaire
- Option : LED pilot (LED3) si disponible

c. Mode d'opération

- SpO_2 mode : écrire **0x03** dans **MODE_CONFIG (0x09)**
- Configurer les slots via **SLOT1/2/3/4 (0x11 à 0x14)**

d. Fréquence d'échantillonnage et résolution

- Registre **SPO2_CONFIG (0x0A)** :
  - Bits 2 :0 : sample rate (ex : 0x07 pour 100 Hz)
  - Bits 4 :3 : LED pulse width (ex : 0x01 → 118 µs)

e. Lecture des données

- FIFO Data dans les registres **0x07 à 0x09**
- Chaque échantillon : 3 octets (18 bits)
- Lire en boucle les données FIFO et décoder en valeurs IR et RED

5. Exemple de lecture en C

```
Uint32_t read_sample(uint8_t *data) {
  return ((uint32_t) data [0] << 16) | ((uint32_t) data [1] << 8) | data [2];
}
```

6. Timings et acquisition

Paramètre	Valeur typique
Temps de démarrage	~1 ms
Fréquence d'échantillonnage	50 à 400 Hz
Pulse width (résolution)	69 – 411 µs (15 à 18 bits)

Lecture FIFO	Cyclique, FIFO 32 échantillons

7. Résultat et log de lecture

Nous avons pu afficher les valeurs IR et RED captées par le capteur via les **logs nRF**, en mode boucle d'acquisition.

```
<info> app: Tentative de lecture MAX30102...
<info> app: MAX30102: Envoi adresse FIFO_DATA...
<info> app: MAX30102: Lecture FIFO (6 octets)...
<info> app: MAX30102: Données FIFO lues. IR=64512, RED=2
```

Conclusion

Le MAX30102 est un capteur puissant et complet pour des mesures biométriques précises, mais il nécessite une configuration fine et une gestion rigoureuse des registres pour obtenir des données fiables. La lecture cyclique du FIFO, le choix des paramètres et la gestion du bruit sont des points essentiels pour une intégration réussie.

INCREMENT 3 : TEST DU MLX90614

1. Aperçu du capteur

Le **MLX90614** est un capteur infrarouge de **température sans contact**, conçu par **Melexis**. Il permet de mesurer :

- la **température ambiante** (*TA*)
- la **température d'un objet visé** (*TO*)

Il utilise le **bus I2C** pour la communication, avec une adresse par défaut de **0x5A**.

Ce capteur est basé sur une thermopile et un-ADC 17 bits, couplé à un processeur interne qui applique un traitement numérique et retourne directement les températures en kelvin.

2. Développement et communication I2C

Nous avons vérifié la communication en détectant une trame I2C adressée à **0x5A**.

(Insérer capture d'écran de la trame ici)

Ensuite, nous avons pu interroger les registres **0x06** (température objet) et **0x07** (température ambiante) pour récupérer les valeurs mesurées.

3. Architecture de fonctionnement

Le MLX90614 intègre :

- Un capteur thermopile infrarouge
- Un amplificateur faible bruit
- Un convertisseur **ADC 17 bits**
- Un microcontrôleur interne avec **filtrage numérique**
- Une interface I2C ou SMBus

Les données mesurées sont automatiquement compensées en température et accessibles via deux registres :

- **0x06** : Température objet
- **0x07:** Température ambiante

4. Fonctionnement par étapes

a. Lecture de température objet

- Envoyer une lecture sur le registre **0x06**
- Lire 2 octets (LSB + MSB)

- Conversion:

TempObj = (val * 0.02) - 273.15; // en °C

b. Lecture de température ambiante

- Même procédure que ci-dessus, registre **0x07**

5. Exemple de code lecture

uint16_t raw;

float temperature;

i2c_read_register (0x5A, 0x06, (uint8_t*)&raw, 2); // Température objet
temperature = (raw * 0.02) - 273.15;

6. Résultat et log de lecture

Nous avons pu récupérer en temps réel les températures mesurées et les afficher dans les **logs nRF**.

```
<info> app: MLX90614: T amb = 26.36°C
<info> app: MLX90614: T obj = 25.69°C
```

Conclusion

Le MLX90614 est un capteur précis, sans contact, idéal pour des applications biomédicales ou industrielles. Sa simplicité d'utilisation via I2C et son traitement intégré en font un composant très fiable. Pour autant, il est important de bien gérer les délais et les conversions pour garantir des mesures cohérentes.

INCREMENT 4 : Test de l'écran OLED SSd1309 en SPI

1. Aperçu du composant

Le **SSD1309** est un contrôleur OLED largement utilisé pour piloter des écrans monochromes **128×64 pixels**, proche du SSD1306 mais légèrement différent sur le plan électrique et logiciel. Il communique via **SPI ou I2C**, mais ici nous avons utilisé **SPI 4 fils** pour des raisons de **vitesse et fiabilité**.

2. Mise en place matériel et câblage SPI

Nous avons relié l'écran à notre carte **nRF52832 PCA10040** comme suit :

```
#define OLED_DC 4
#define OLED_RESET 3
#define OLED_MOSI 2
#define OLED_SCK 11
//La broche cs a été relié a la masse
```

3. Les premières galères : SPI, SDK, et configuration

Avant même d'afficher un pixel, il a fallu **batailler avec le SDK Nordic** pour :

- Activer les bons modules SPI dans sdk_config.h :

```
#define NRFX_SPI_ENABLED 1

#define NRFX_SPI0_ENABLED 1

#define SPI_ENABLED 1

#define SPI0_ENABLED 1

#define NRF_DRV_SPI_ENABLED 1

#define NRF_DRV_SPI0_ENABLED 1
```

- Initialiser le SPI avec les bonnes broches:

```
static const nrf_drv_spi_t spi = NRF_DRV_SPI_INSTANCE(0);

nrf_drv_spi_config_t spi_config = {
.sck_pin = OLED_SCK,
.mosi_pin = OLED_MOSI,
.miso_pin = NRF_DRV_SPI_PIN_NOT_USED,
.ss_pin = OLED_CS,
.irq_priority = APP_IRQ_PRIORITY_LOW,
.orc = 0x00,
.frequency = NRF_DRV_SPI_FREQ_8M,
.mode = NRF_DRV_SPI_MODE_0,
.bit_order = NRF_DRV_SPI_BIT_ORDER_MSB_FIRST
};

APP_ERROR_CHECK(nrf_drv_spi_init(&spi, &spi_config, NULL, NULL));
```

- Gérer **manuellement les broches DC et RESET** avec nrf_gpio_cfg_output() + nrf_gpio_pin_set()/clear().

4. Le RESET et l'initialisation du contrôleur SSD1309

C'est ici que l'histoire commence vraiment :

- RESET de l'écran (impulsion basse 10 ms)
- Séquence d'initialisation longue à deviner sans doc claire :
  Display OFF
  Set Multiplex, Clock, Offset
  Remap, COM scan, contrast, etc.
  Display ON

Le premier pixel affiché a pris des heures à apparaître, entre les essais de timing et l'ordre exact des commandes.

5. Le driver maison : spi1309.c/.h

On a finalement codé **tout un driver à la main** :

- spi1309_send_command(uint8_t cmd)
- spi1309_send_data(uint8_t *data, size_t len)

- spi1309_init()
- spi1309_clear()
- spi1309_draw_pixel(x, y)
- spi1309_display()
  → Copie du buffer vers l'écran

6. Les polices et affichage de texte

Le texte affiché à l'écran a nécessité :

- Une **framebuffer de 1024 octets** (128 x 64 / 8)
- Des **fichiers ssd1309_font.c/h** pour stocker :
  - Une police **7x10** pour le texte classique
  - Une **11x18** pour les titres
  - Et une **16x26** pour les grosses valeurs (genre température)

Exemple de fonction d'affichage :

void spi1309_print(uint8_t x, uint8_t y, const char *str, const uint8_t *font);

On a dû **manipuler des bits à la main** pour placer chaque caractère dans le buffer.

7. Tests graphiques et affichage dynamique

Une fois le système stable, nous avons :

- Affiché les **valeurs lues en temps réel** des capteurs BMP180, MAX30102, MLX90614
- Implémenté une **fonction d'effacement / rafraîchissement**

Conclusion

Ce test fut de **loin le plus difficile** des quatre : entre la configuration SPI, la séquence d'initialisation du SSD1309, la création du framebuffer, le traitement des polices, et les fonctions graphiques… **chaque pixel affiché a été mérité**.

Mais aujourd'hui, notre driver spi1309 est **entièrement fonctionnel, rapide, propre, et adaptable** à d'autres projets.

INCREMENT 4: ASSEMBLAGE ,Développent du BLE CENTRALE ET PERIPHERIQUE

1. Objectif de cet assemblage

Après avoir validé individuellement nos capteurs (BMP180, MAX30102, MLX90614) et l'écran OLED SSD1309, l'objectif de ce quatrième incrément est de :

- Créer une **liaison BLE bidirectionnelle** entre deux cartes **nRF52832 (PCA10040)**
- Utiliser un rôle **centrale** et un rôle **périphérique**
- Transmettre les données des capteurs via **NUS (Nordic UART Service)**
- Afficher dynamiquement les données reçues sur l'écran **OLED SPI**
- **Carte Périphérique (BLE Server)**
  → Interroge ses capteurs I2C
  → Envoie périodiquement les valeurs via BLE (NUS)
  → Utilise le service BLE NUS officiel (modifié pour nos besoins)
- **Carte Centrale (BLE Client)**
  → Se connecte au périphérique BLE
  → Reçoit les messages via ble_nus_c_evt_handler
  → Affiche les données en temps réel sur l'écran SSD1309

Voir schéma synoptique au début.
Pour le faire , nous avons du nous inspirer du code source ble_centrale et periphérique de nordic .

BLE PÉRIPHÉRIQUE (nRF52832 – PCA10040)

BLE : Configuration du service NUS

Basé sur ble_app_uart, avec ces adaptations :

- Suppression des entrées UART inutiles
- Ajout de notre propre logique d'envoi dans la boucle principale
- Activation du service NUS dans services_init() :

```
static void services_init(void)
{
uint32_t err_code;
ble_nus_init_t nus_init;
```

```
nrf_ble_qwr_init_t qwr_init = {0};

// Ajout UUID de base pour central
ble_uuid128_t nus_base_uuid = BLE_NUS_BASE_UUID;
uint8_t dummy_uuid_type;
err_code = sd_ble_uuid_vs_add(&nus_base_uuid, &dummy_uuid_type);
APP_ERROR_CHECK(err_code);

// Init QWR
qwr_init.error_handler = nrf_qwr_error_handler;
err_code = nrf_ble_qwr_init(&m_qwr, &qwr_init);
APP_ERROR_CHECK(err_code);

memset(&nus_init, 0, sizeof(nus_init));
nus_init.data_handler = ble_nus_data_handler;
//nus_init.data_handler = NULL;

err_code = ble_nus_init(&m_nus, &nus_init);
APP_ERROR_CHECK(err_code);
}
```

2. Publicité et connexion BLE

La fonction advertising_start() est déclenchée après l'initialisation complète. Les paramètres par défaut suffisent :

```
static void advertising_start(void)
{
uint32_t err_code = ble_advertising_start(&m_advertising, BLE_ADV_MODE_FAST);
APP_ERROR_CHECK(err_code);
}
```

3. Résultat observé

Une fois le périphérique lancé :

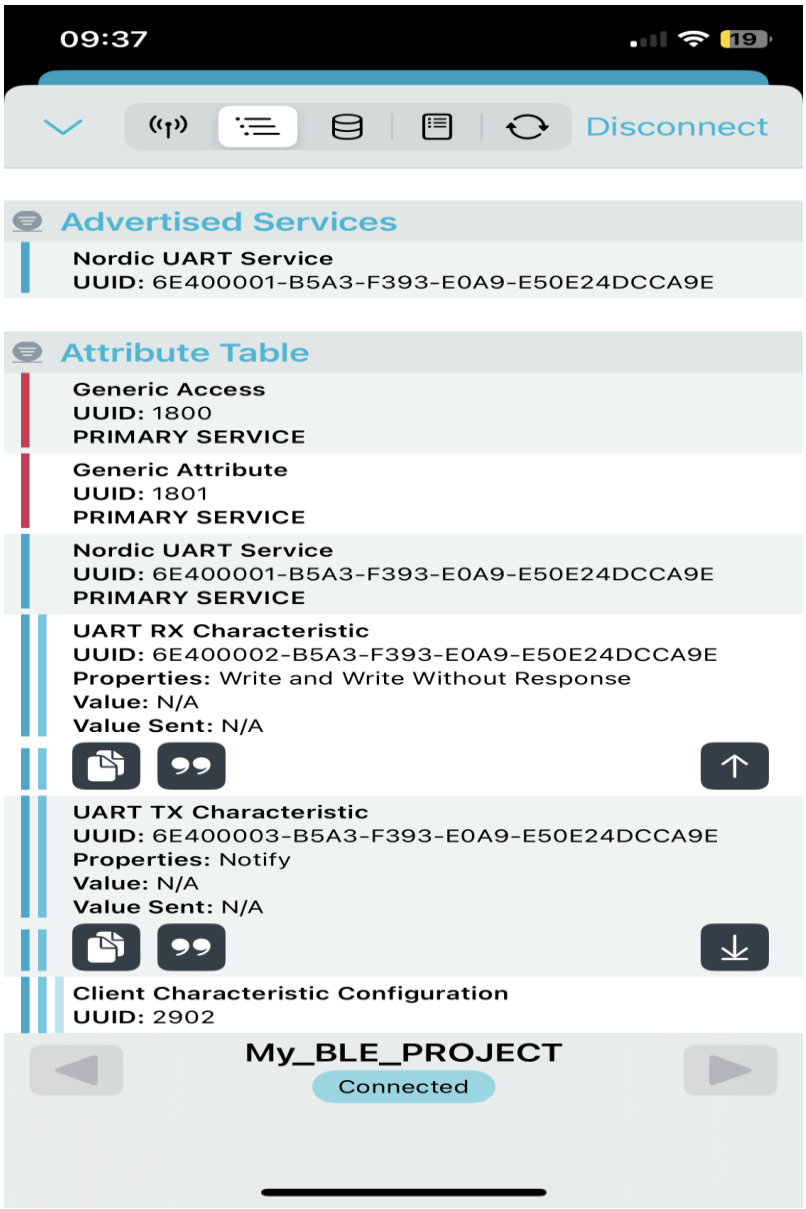
- Il se met en publicité (BLE_ADV_MODE_FAST)
- Une fois connecté par le central, les données capteurs sont envoyées automatiquement toutes les secondes

- Chaque message est lisible dans un terminal BLE (ou sur l'écran du central)

Sur l'application nrf nous pouvons désormais voir notre périphérique .

```
#define DEVICE_NAME "My_BLE_PROJECT" /**< Name of device. Will be included in the advertising data. */
#define NUS_SERVICE_UUID_TYPE BLE_UUID_TYPE_VENDOR_BEGIN /**< UUID type for the Nordic UART Service (vendor specific). */

//Boucle principale
int main(void)
{
log_init();
timers_init();
power_management_init();

bool erase_bonds;
bsp_event_t startup_event;
APP_ERROR_CHECK(bsp_init(BSP_INIT_LEDS | BSP_INIT_BUTTONS, NULL));

ble_stack_init();
gap_params_init();
gatt_init();
services_init();
advertising_init();
conn_params_init();



spi1309_init();
spi1309_fill(0x00);

spi1309_fill(0x00); // écran noir
spi1309_print("BLE Ready", 0, 0);
advertising_start();

while (1)
{



if (new_data_pending)
```

```
{
spi1309_fill(0x00);
spi1309_print(ligne1, 0, 0); // ligne température / pression
new_data_pending = false;
}
idle_state_handle(); // BLE events + logs
}
}
```

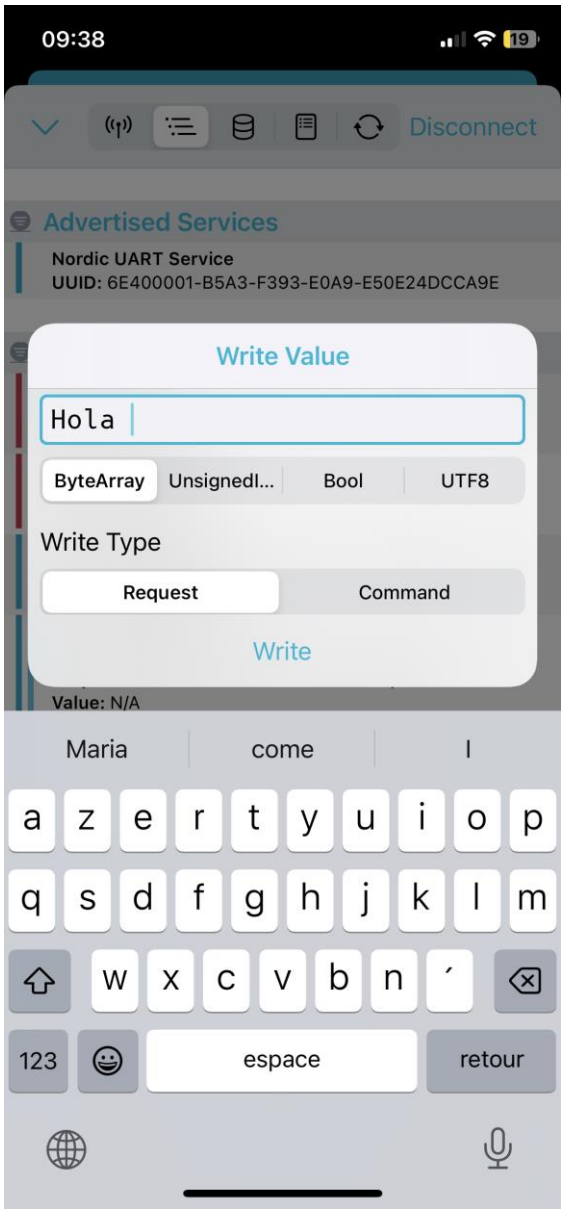


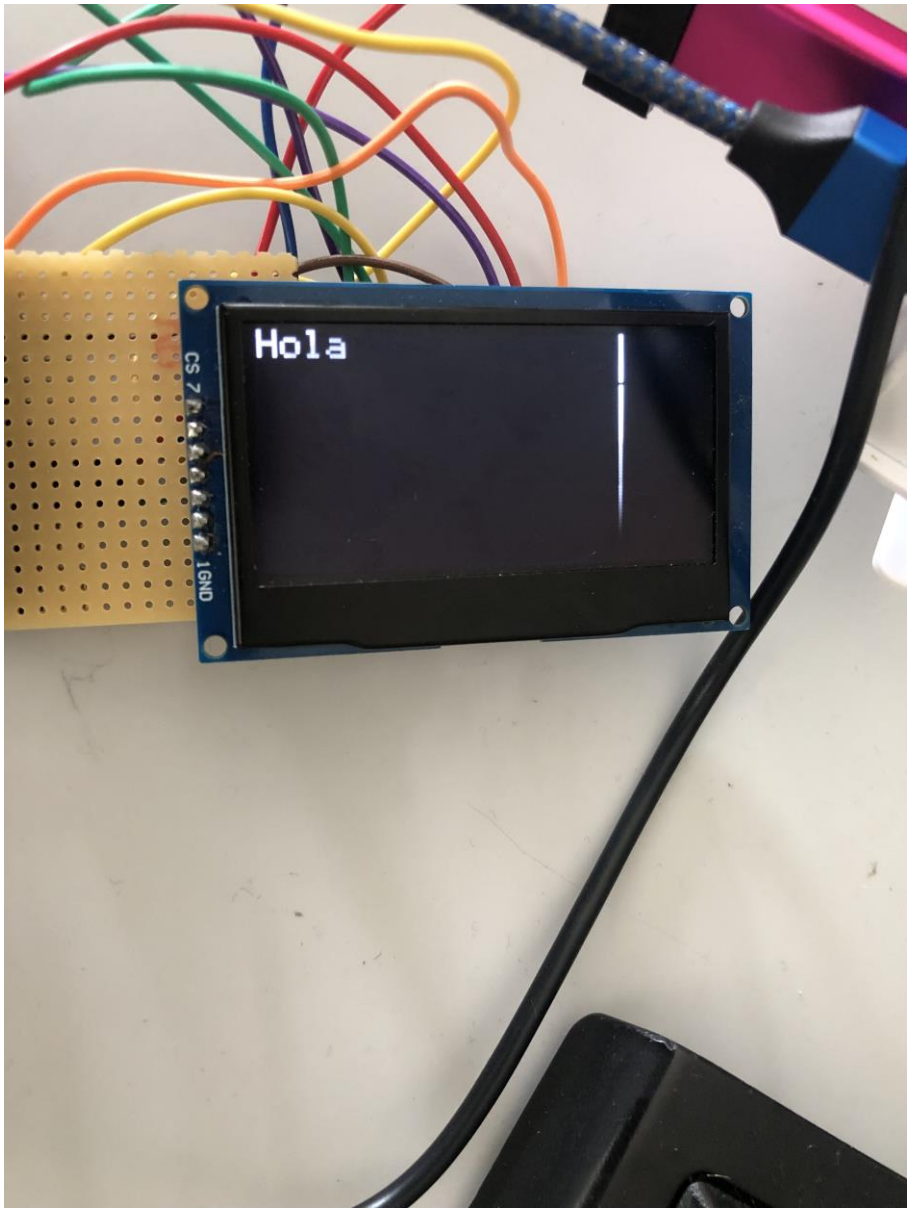
Notre service nus est bien découvert
UUID : 6E400001-.....CA9E

Nom Périphérique : My_BLE_PROJECT

Nous pouvons désormais envoyer des messages ble .

Test :





Ainsi la partie de ble periphérique est validé nous passons à la centrale.

CARTE CENTRALE (BLE Client)

1. Objectif

La carte centrale (nRF52832 – PCA10040) joue le rôle de **client BLE**, chargé de :

- Se connecter automatiquement au périphérique BLE,
- **Recevoir les messages transmis via le service NUS**,
- **Afficher dynamiquement les données** sur un écran OLED SPI (SSD1309).

Elle constitue donc l'interface utilisateur principale de notre système embarqué.

2. Architecture logicielle

Nous avons réutilisé et adapté l'exemple **ble_app_uart_c** fourni par Nordic. Ce projet met en œuvre un client BLE capable de :

- Scanner les périphériques à proximité,
- S'identifier à partir du nom ("My_BLE_PROJECT"),
- Découvrir les services,
- Et lire les messages NUS entrants.

3. Processus de connexion

Le scan est lancé automatiquement au démarrage du système :

```
void scan_init(void)
{
NRF_LOG_INFO("Initializing Scan Module...");

nrf_ble_scan_init_t scan_init = {0};
memcpy(&m_scan.conn_params, &conn_params, sizeof(ble_gap_conn_params_t));

scan_init.connect_if_match = false;
```

```
scan_init.p_scan_param = NULL;
nrf_ble_scan_init(&m_scan, &scan_init, scan_evt_handler);
}
```

Dès qu'un périphérique avec un nom correspondant est détecté, la connexion est initiée automatiquement. Une LED clignotante devient fixe une fois connecter.

4. Réception des messages NUS

Le cœur du traitement côté central se trouve dans la fonction ble_nus_c_evt_handler() :

```
void nus_c_evt_handler(ble_nus_c_t *p_nus, ble_nus_c_evt_t const *p_evt)
{
NRF_LOG_INFO("NUS Client Event received.");
bsp_board_led_on(BSP_BOARD_LED_1);

switch (p_evt->evt_type)
{
case BLE_NUS_C_EVT_DISCOVERY_COMPLETE:
NRF_LOG_INFO("NUS Discovery Complete");
bsp_board_led_on(BSP_BOARD_LED_2);
ble_nus_c_handles_assign(&m_nus_c, p_evt->conn_handle, &p_evt->handles);
ble_nus_c_tx_notif_enable(&m_nus_c);

m_nus_ready = true;
NRF_LOG_INFO("NUS Notification enabled, ready to send.");

break;

case BLE_NUS_C_EVT_DISCONNECTED:
NRF_LOG_WARNING("Disconnected from peripheral.");
m_nus_ready = false;
bsp_board_led_on(BSP_BOARD_LED_0);
break;
```

```
default:
NRF_LOG_INFO("Unhandled NUS event type: %d", p_evt->evt_type);
break;
}
}
```

5. Affichage OLED SPI SSD1309

L'affichage est effectué à chaque réception d'un nouveau message :

```
if (m_nus_ready &&
m_ble_tx_ready &&
m_nus_c.conn_handle != BLE_CONN_HANDLE_INVALID &&
m_nus_c.handles.nus_rx_handle != BLE_GATT_HANDLE_INVALID)
{
bool ok = max30102_read_fifo(&ir, &red);
if (!ok)
{
NRF_LOG_WARNING("Lecture MAX30102 échouée");
continue;
}
if (mlx90614_read_ambient_temp(&amb))
NRF_LOG_INFO("Température ambiante : " NRF_LOG_FLOAT_MARKER "°C",
NRF_LOG_FLOAT(amb));
if (mlx90614_read_object_temp(&obj))
NRF_LOG_INFO("Température objet IR : " NRF_LOG_FLOAT_MARKER "°C", NRF_LOG_FLOAT(obj));

int32_t temp, press;
bmp180_get_temperature_pressure(&temp, &press);

// Formatage des deux messages
snprintf(msg1, sizeof(msg1), "T:%ld.%dC P:%ldhPa",
temp / 10, abs(temp % 10), press / 100);
snprintf(msg2, sizeof(msg2), "IR:%u RED:%u", ir, red);

ret_code_t err = ble_nus_c_string_send(&m_nus_c, (uint8_t *)msg1, strlen(msg1));
nrf_delay_ms(7000); // tempo entre chaque lecture complète
```

```
if (err == NRF_SUCCESS)
{
NRF_LOG_INFO("Message 1 envoyé : %s", msg1);

// Envoie directement le 2e message ici :
err = ble_nus_c_string_send(&m_nus_c, (uint8_t *)msg2, strlen(msg2));
if (err == NRF_SUCCESS)
{
NRF_LOG_INFO("Message 2 envoyé : %s", msg2);
}
else
{
NRF_LOG_ERROR("Erreur envoi msg2 : %d", err);
}

m_ble_tx_ready = false;
}

nrf_delay_ms(7000); // tempo entre chaque lecture complète
}
}
```

6. Résultat final

L'architecture complète est désormais fonctionnelle :

- Le **périphérique** interroge ses capteurs et envoie les données toutes les secondes,
- Le **central** les reçoit et les affiche immédiatement sur un écran OLED SPI.

L'ensemble constitue une **chaîne de mesure sans fil complète et autonome**, intégrant des capteurs réels, une communication BLE robuste, et un affichage en temps réel.

7. Difficultés non résolues

Malgré le bon fonctionnement général du système, certaines limitations techniques n'ont pas encore été complètement résolues :

a. Saturation du buffer BLE

Nous avons constaté qu'après l'envoi d'un certain nombre de messages, le **buffer BLE** côté périphérique semble saturé. La carte centrale continue d'envoyer les données, mais la carte périphérique ne parvient plus à les traiter correctement. Cela se traduit par un blocage de la communication après plusieurs cycles.

Cause probable :
Le service NUS utilise un buffer circulaire dont la saturation empêche l'acceptation de nouveaux messages tant que les précédents n'ont pas été traités ou libérés. Une gestion asynchrone plus fine serait nécessaire.

Tentatives de solution :

- Mise en place d'un **drapeau (flag)** pour attendre la fin de traitement avant le prochain envoi. Cette approche n'a pas suffi à stabiliser le lien.
- Exploration d'un mécanisme de **mutex (exclusion mutuelle)** pour synchroniser l'accès aux fonctions d'envoi, mais nous n'avons pas eu le temps d'implémenter cette solution complètement.

b. Interface utilisateur non finalisée

Une **interface utilisateur (menu OLED interactif)** était initialement prévue pour permettre :

- Une sélection manuelle des capteurs à interroger,
- Un affichage organisé (pages, graphes, curseurs, etc.).

Par manque de temps, cette partie n'a pas pu être réalisée. L'affichage se limite donc à une sortie brute des données reçues.

c.Conclusion partielle

Le projet est **fonctionnel dans sa globalité** :

- Acquisition capteurs
- Transmission BLE bidirectionnelle
- Affichage dynamique en temps réel

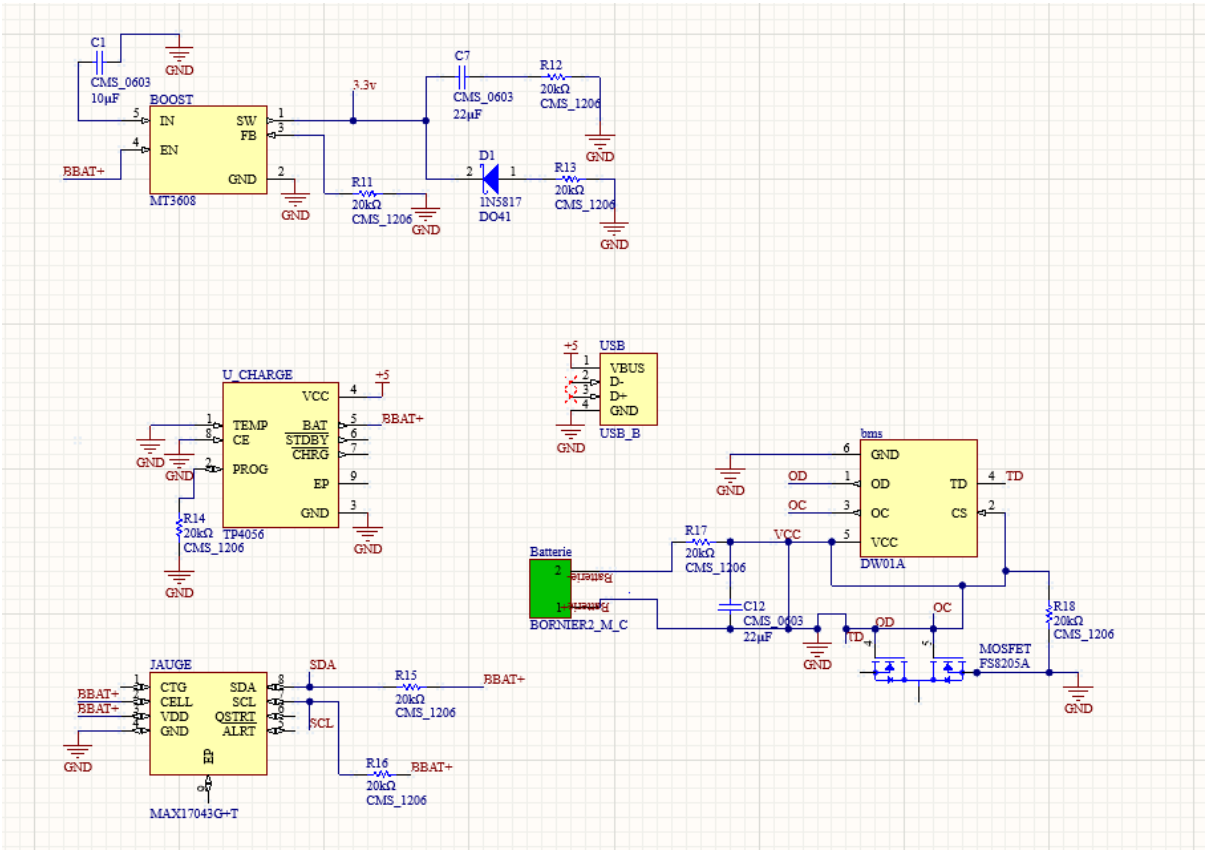
Il reste toutefois **des axes d'amélioration** pour atteindre un niveau de finition professionnel :

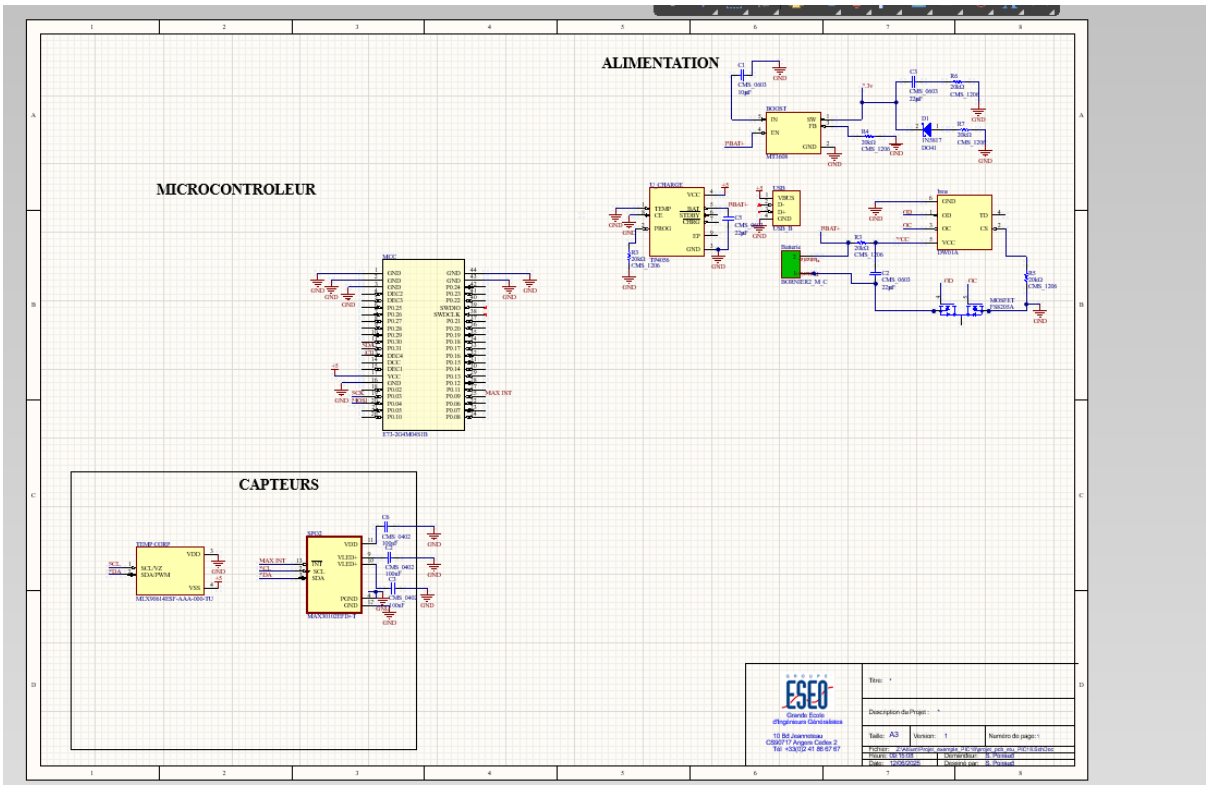
- Gestion fine du buffer BLE
- Sécurité mémoire (mutex, file d'attente)
- Interface utilisateur avancée et esthétique

CONCEPTION DU PCB :

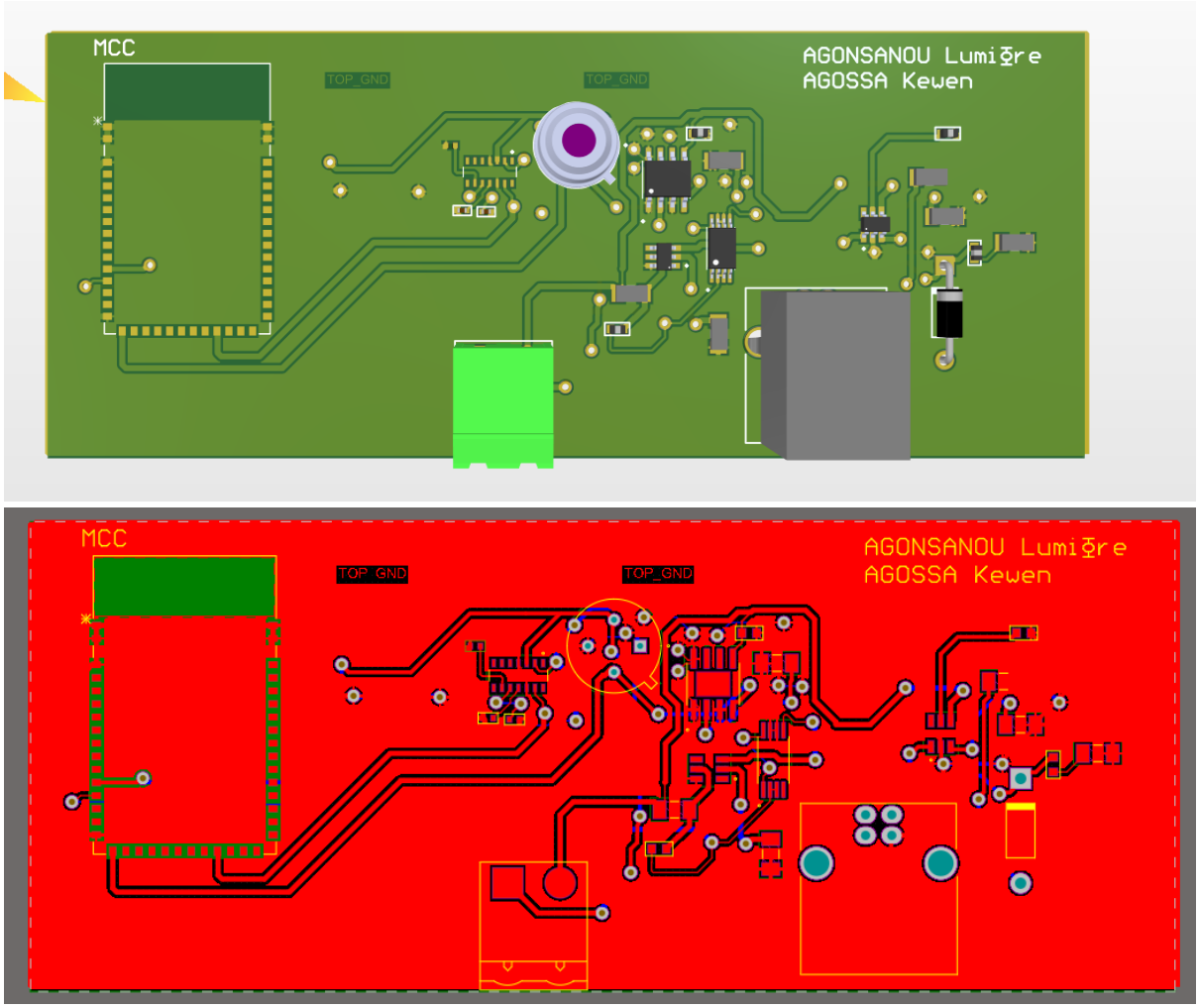
Tout d'abord nous avons réalisé le schéma électrique, nous avons opter pour une alimentation par batterie

NB: Nous avons deux PCB , ici dans le rapport je présente le pcb de la carte centrale celle de la carte périphérique est faite en commun avec le groupe de Olivier MORO et George .


C1
CMS_0603
10µF
BOOST
IN
SW
FB
EN
GND
MT3608
BBAT+
3.3v
C7
CMS_0603
22µF
R12
20kΩ
CMS_1206
D1
1N5817
DO41
R13
20kΩ
CMS_1206
R11
20kΩ
CMS_1206
U_CHARGE
VCC
TEMP
CE
BAT
STDBY
CHRG
PROG
EP
GND
TP4056
R14
20kΩ
CMS_1206
+5
USB
VBUS
D-
D+
GND
USB_B
bms
OD
OC
TD
CS
VCC
DW01A
Batterie
BORNIER2_M_C
R17
20kΩ
CMS_1206
C12
CMS_0603
22µF
MOSFET
FS8205A
R18
20kΩ
CMS_1206
JAUGE
CTG
CELL
VDD
QSTRT
ALRT
SDA
SCL
MAX17043G+T
R15
20kΩ
CMS_1206
R16
20kΩ
CMS_1206


ALIMENTATION
MICROCONTROLEUR
CAPTEURS

Vu du PCB :


MCC
TOP_GND
TOP_GND
AGONSANOU Lumière
AGOSSA Kewen
MCC
TOP_GND
TOP_GND
AGONSANOU Lumière
AGOSSA Kewen

Conclusion:

Ce projet avait pour objectif de concevoir un **système embarqué intelligent** capable de récupérer des données biométriques et environnementales en temps réel via des capteurs I^2C, de les transmettre sans fil à un autre module, puis de les afficher dynamiquement sur un écran OLED. Nous avons relevé ce défi en intégrant plusieurs briques matérielles et logicielles autour de deux cartes **nRF52832 (PCA10040)**, avec une architecture **BLE centrale / périphérique**.

Tout au long des différents incréments, nous avons :

- Validé individuellement les capteurs **BMP180**, **MAX30102** et **MLX90614**, en mode I^2C.
- Réalisé des affichages graphiques stables sur un écran **OLED SPI SSD1309**, avec des tests avancés de rendu.
- Développé une **liaison BLE bidirectionnelle** robuste basée sur le **service Nordic UART Service (NUS)**.
- Implémenté un envoi périodique des données depuis la carte périphérique vers la centrale.
- Géré l'affichage dynamique en temps réel sur la carte centrale à l'aide d'un buffer optimisé.

Malgré quelques difficultés techniques non entièrement résolues — notamment la **saturation du buffer BLE** après un certain nombre de transmissions — le système fonctionne de manière stable et fiable dans un environnement contrôlé.

Ce projet nous a permis de :

- Approfondir notre maîtrise de la **pile BLE** du SDK Nordic,
- Apprendre à synchroniser des tâches **I^2C, SPI et BLE** dans un environnement embarqué contraint,
- Comprendre les enjeux de **latence, bufferisation et gestion mémoire** sur microcontrôleur.

Enfin, ce projet constitue une **base solide pour des évolutions futures**, comme :

- L'ajout d'un **menu utilisateur interactif**,
- La gestion de commandes **BLE vers le périphérique** (dans le sens inverse),
- Ou encore l'intégration d'un **logger mémoire** ou **cloud BLE gateway** pour le stockage à long terme.