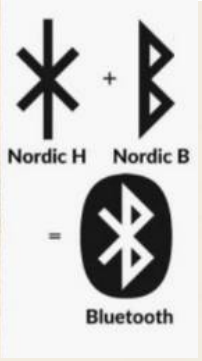
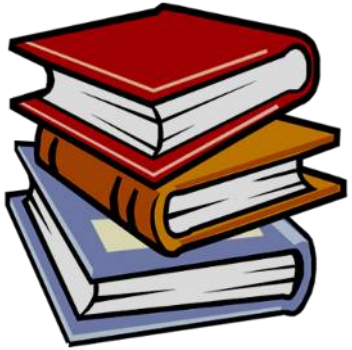


Bluetooth Low Energy & ESP32-C3

Bluetooth®

Felhasznált és ajánlott irodalom

- **Leírások, dokumentáció**
  - Adafruit: Introduction to Bluetooth Low Energy
  - Neil Kolban: Kolban's book on ESP32
  - Neil Kolban: BLE C++ Guide.pdf
  - ESPRESSIF: ESP32 Arduino Core Documentation



- **Szoftver segédlet**
  - Version 4 UUID Generator
  - Nordic Semiconductor ASA: nRF Connect for Mobile
  - Ryan Powell: NimBLE-Arduino library
  - Adafruit: BME280 sensor library

Mi a Bluetooth® ?

- A **Bluetooth** rövid hatótávolságú, adatcseréhez használt, nyílt, vezetéknélküli kommunikációs szabvány
- Alkalmazásával számítógépek, mobiltelefonok, fülhallgatók és egyéb készülékek között kis hatótávolságú rádiós kapcsolatot létesíthetünk a szabadon elérhető 2,4 gigahertzes frekvenciasávban
- A név a lázongó dán, norvég és svéd törzseket egyesítő **Harald Blåtand** dán király nevének angol változata, aki nagyon szerette az áfonyát, ezért kék lett a foga. A **Bluetooth**-t is arra szánták, hogy különböző eszközöket egyesítsen és összekössön
- A „klasszikus” **Bluetooth** mód mellett a 2010-ben megjelent 4.0 specifikációtól kezdve lehetőség van a **BLE** (Bluetooth Low Energy) üzemmód használatára is, ami elsősorban az energiatakarékos IOT eszközök szempontjából fontos előrelépés
- Az **ESP32-C3** mikrovezérlő **Bluetooth LE 5.0** támogatással rendelkezik

BLUETOOTH PICONET TOPOLÓGIA: KÖZPONT ÉS PERIFÉRIÁK

A Bluetooth kommunikáció egy mester és egy szolga csomópont között történhet

Minden állomás egy 48 bites rögzített eszköz-címhez van társítva

Az eszközcímekhez egy szimbolikus név (display name) is társítható, ami nem egyedi

Klasszikus Bluetooth (BR/EDR)

- Folytonos kapcsolat, nagy sávszélesség (1–3 Mbit/s), magas fogyasztás.
- Hangátvitelre és perifériákra tervezve.

Bluetooth Low Energy (BLE)

- Szaggatott, rövid rádiós aktivitás.
- Mikroamperes fogyasztás, villámgyors kapcsolatfelépítés.
- Szenzorokhoz és IoT-hez tervezve.

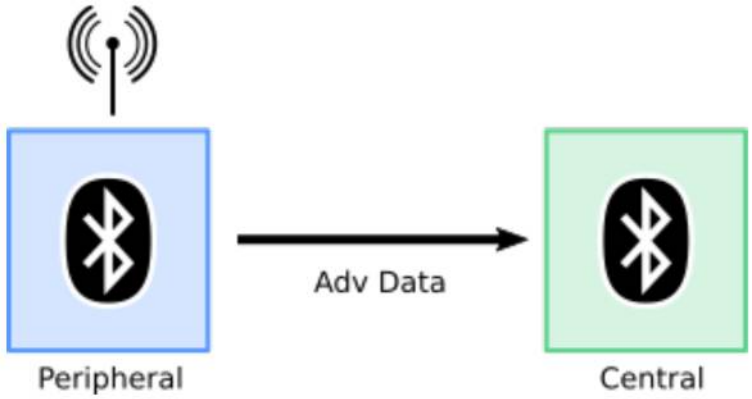
A Paradigma: “Nem a sávszélesség a lényeg, hanem az alvásidő.”

Bluetooth profilok és protokollok

- Az alsóbb szinteken a **Bluetooth** a partnerek közötti adatcseréről gondoskodik. A **Bluetooth** azonban sokkal többet jelent, mint egyszerű adatcsere: egységes keretbe foglalja és értelmezi az adatokat
- A több gyártó által épített eszközök közötti együttműködés érdekében magasabb szintű profilokat határoztak meg. Ezek írják le, hogy milyen szabványosított szolgáltatásokkal valósítható meg egy adott eszközfunkció. Fontos tudni, hogy a **klasszikus Bluetooth** és a **BLE** profiljai eltérő felépítésűek, de ebben az előadásban csak az utóbbival foglalkozunk
- Néhány **BLE** profil, amellyel gyakran találkozhatunk:
  - **HRP** – Heart Rate Profile (pl. pulzusmérő mellkaspántok)
  - **CSCP** – Cycling Speed and Cadence Profile (pl. kerékpár szenzorok)
  - **HOGP** – HID over GATT Profile (pl. energiatakarékos egér vagy billentyűzet)
  - **ESP** – Environmental Sensing Profile (pl. hőmérséklet és páratartalom mérő)
  - **BLP** – Blood Pressure Profile (pl. digitális vérnyomásmérők)
  - **FMP** – Find Me Profile (pl. kulcskereső és elvesztésgátló eszközök)

BLE: Hogyan találnak egymásra az eszközök?

- **GAP** (Generic Access Protocol) – az általános hozzáférési protokoll határozza meg a felderítési folyamatot, az eszközkezelést, valamint a **BLE** eszközök közötti eszközkapcsolat kialakítását
- A **GAP** szempontjából az eszközöknek két osztálya van: a központi eszköz és a periféria
- A perifériák hirdethetik magukat (**advertising**), illetve pásztázáskor (**scan**) válaszüzenetet küldhetnek. A hirdetés 20 ms – 10.24 s időközönként történhet, az üzenet legfeljebb 31 bájt adatot tartalmazhat
- A hirdetés egyfajta *broadcast* üzenetként is felfogható, s az üzenetcsomag nyilvánosan sugárzott adatot is tartalmazhat, így akár kapcsolódás nélkül is továbbíthatunk adatot (egyirányú adatküldés)
- **Nézzük meg ezt a gyakorlatban!** Készítünk egy programot, amely kapcsolódás nélkül, közvetlenül a hirdetési csomagban küldi el a szenzoradatokat



1. Projekt: "Kiáltás a sötétbe" (Broadcast-only)

A feladat: Hőmérséklet sugárzása kapcsolatfelépítés nélkül. Ez a BLE "Hello World" alkalmazása.

A 31-bájtos Reklámcsomag szerkezete:

(Opcionális név)

Flags: 0x06
AD Típus: 0x01
(Általánosan felfedezhető)

Name: C3-BME280
AD Típus: 0x09
(Complete Local Name)

AD Típus: 0xFF **Payload:**
Manufacturer Specific Data
(Gyártói ID (2 bájt) + Hőmérséklet tizedfokokban (2 bájt hex))

3 bájt | 11 bájt | 6 bájt

Három Advertising Data blokkot küldünk ki

Összesen felhasznált: 20/31 bájt (11 bájt szabad)

A Reklámcsomag Összeállítása (ESP32-C3 / C++)

```
// Hőmérséklet tizedfokban (pl. 23.6 C -> 236 -> 0x00EC)
int16_t t = (int16_t)(tempC * 10);
uint8_t mfgData[4];
mfgData[0] = 0xFF; // Gyártó ID Low byte
mfgData[1] = 0xFF; // Gyártó ID High byte
mfgData[2] = t & 0xFF;
mfgData[3] = (t >> 8) & 0xFF;

NimBLEAdvertisementData advData;
advData.setFlags(0x06);
advData.setName("C3-BME280");
advData.addManufacturerData(std::string((char*)mfgData, 4));
pAdvertising->setAdvertisementData(advData);
```

Flags:
00000110 = 0x06
- LE Limited Discoverable Mode
- LE General Discoverable Mode
- BR/EDR Not Supported
- LE and BR/ERD Capable (Controller)
- LE and BR/ERD Capable (Host)
- Reserved

FF FF EC 00 -> 23.6 °C

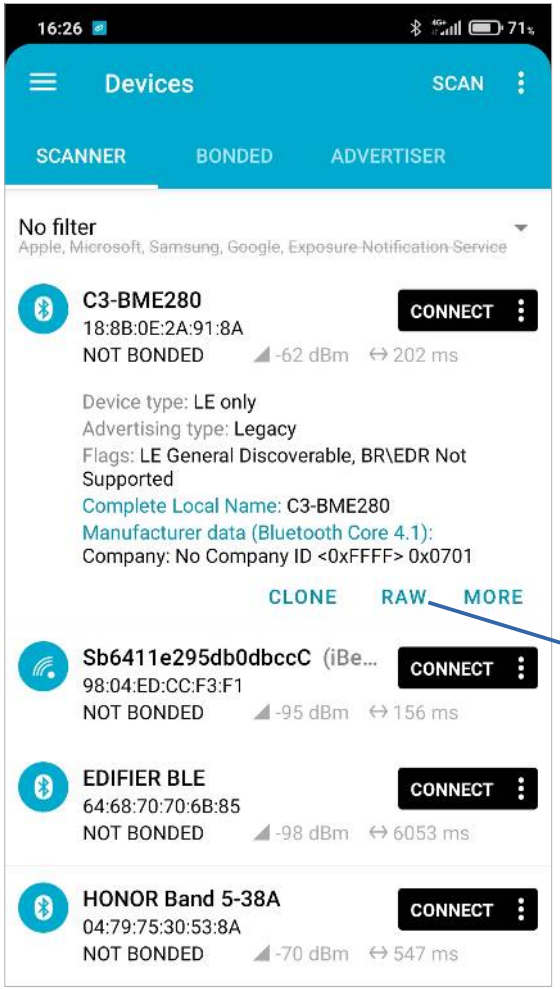
ble_advertising.ino – 2/1.

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>
#include <NimBLEDevice.h>             // BLE kezelő könyvtár
#include <WiFi.h>                     // Szükséges a Wi-Fi lekapcsolásához
#define I2C_SDA 8
#define I2C_SCL 9
Adafruit_BME280 bme;
NimBLEAdvertising *pAdvertising;
unsigned long lastMillis = 0;
const unsigned long interval = 3000;

void setup() {
  Serial.begin(115200);
  WiFi.mode(WIFI_OFF);                // WiFi kikapcsolása
  Wire.begin(I2C_SDA, I2C_SCL);       // Szenzor inicalizálása
  bme.begin(0x76, &Wire))
  NimBLEDevice::init("C3-BME280");    // --- 3. BLE inicializálása ---
  pAdvertising = NimBLEDevice::getAdvertising();
  pAdvertising->setMinInterval(160);
  pAdvertising->setMaxInterval(320);
  Serial.println("Rádió készen áll...");
}
```

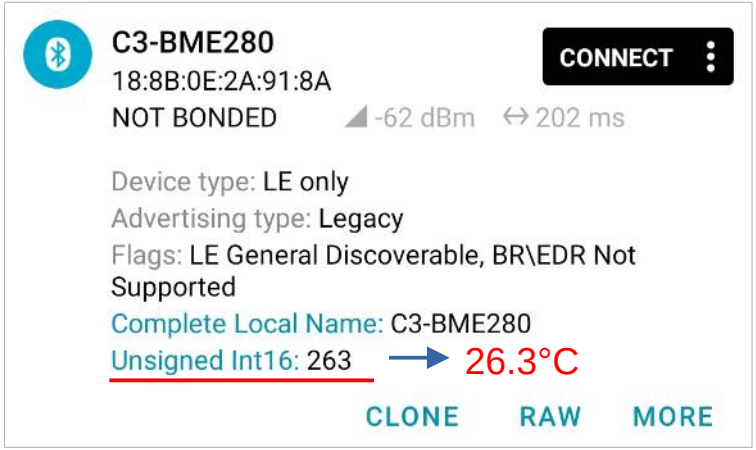
ble_advertising.ino – 2/2.

```
void loop() {
  if (millis() - lastMillis >= interval) {
    lastMillis = millis();
    float tempC = bme.readTemperature();
    if (!isnan(tempC)) {
      int16_t t = (int16_t)(tempC * 10);

      // --- BLE Adatcsomag felépítése ---
      NimBLEAdvertisementData advData;
      advData.setFlags(0x06);              // General Discoverable, LE only
      advData.setName("C3-BME280");        // Complete local name
      uint8_t mfgData[4];                  // Manufacture Specific Data
      mfgData[0] = 0xFF;                   // „No Company” ID
      mfgData[1] = 0xFF;
      mfgData[2] = (uint8_t)(t & 0xFF);    // Hőmérséklet (low endian)
      mfgData[3] = (uint8_t)((t >> 8) & 0xFF);
      advData.setManufacturerData(std::string((char*)mfgData, 4));
      pAdvertising->setAdvertisementData(advData);
      pAdvertising->start();               // Indítás/Frissítés
      Serial.printf("Hőmérséklet: %.1f °C\n", tempC);
    }
  }
}
```

A hirdetési csomagok elemzése



Az **ESP32-C3** által küldött üzeneteket az nRF Connect for Mobile alkalmazással ellenőriztük. Felismerhető a **Manufacturer Specific Data** szakasz, benne s benne a hőmérséklet-adat is (tizedfokokban)



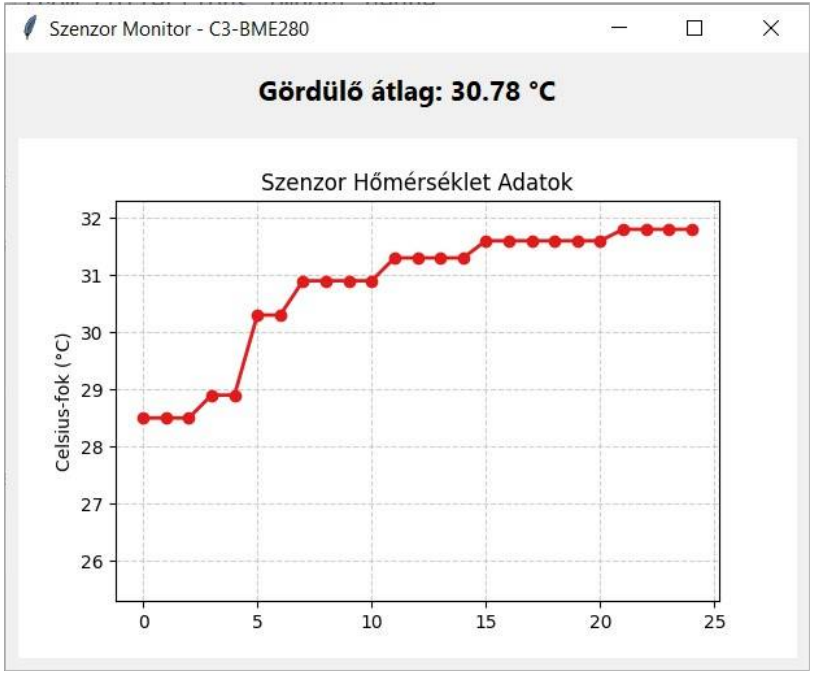
A 0x0701 adat Low Endian alak, tehát valójában 1*256 + 7*1 = 263 az értéke

Ez tizedfokokban értendő, tehát 26.3°C-ot jelent

LEN	TYPE	VALUE
2	0x01	0x06
10	0x09	0x43332D424D45323830
5	0xFF	0xFFFF0701

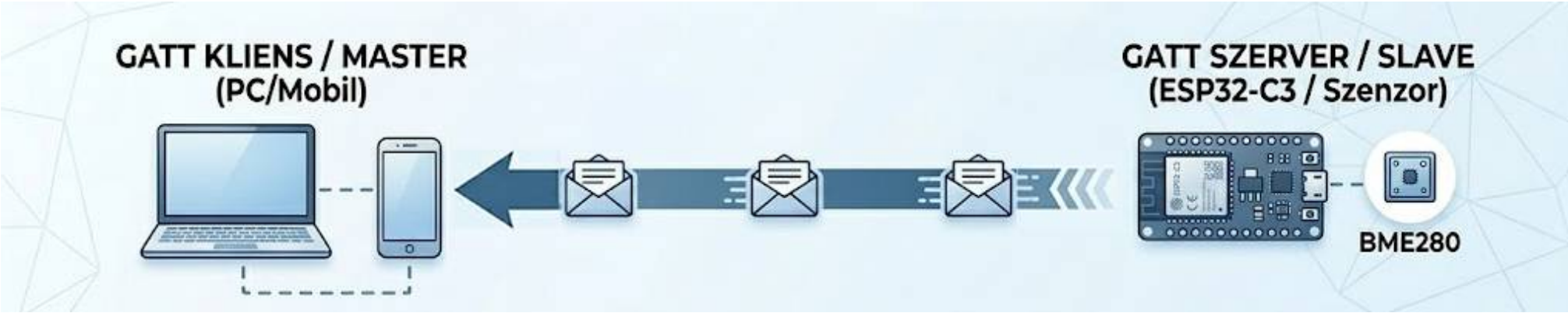
ble_demo.py

- Ez a **Python** alkalmazás a **Bleak** könyvtár segítségével kezeli a **BLE** perifériát. Az észlelt hirdetési csomagokat a *local_name* tulajdonság alapján szűri („C3-BME280”). Csak a sikeres névazonosítás után engedi tovább az adatot a gyártói mező (Manufacturer Data) kibontásához, amiből dekódolja a hőmérsékleti adatot
- A kód három pillérre épül:
  - **Bleak:** Aszinkron BLE könyvtár
  - **Tkinter** és **Matplotlib:** A grafikus felület és a valós idejű grafikon motorja.
  - **Threading:** többszálú futtatás
- Miért kell a többszálúság?
  - A **Tkinter**-nek folyamatosan futnia kell a főszálon (root.mainloop())
  - A **BLE** adatok fogadása aszinkron módon



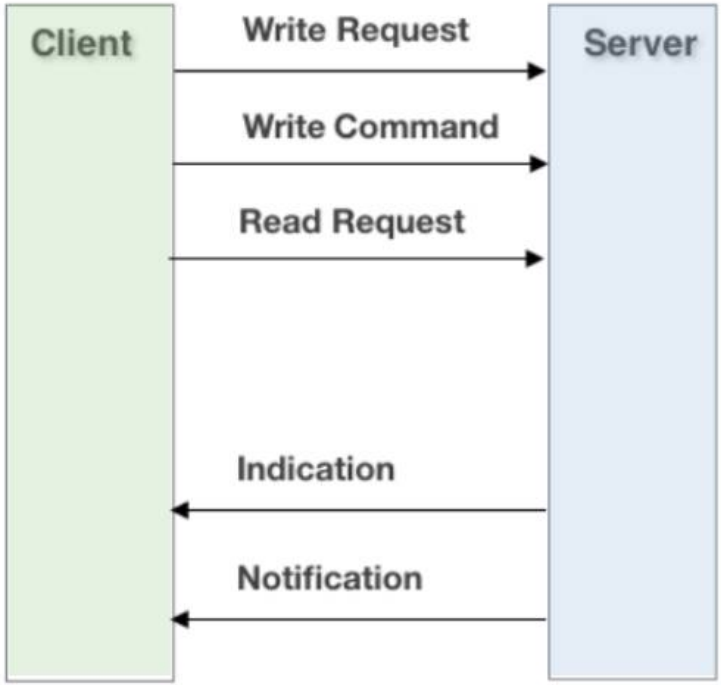
BLE: Hogyan zajlik az adatcsere? (GATT)

- Az Általános Attribútum Protokoll (**GATT**) mechanizmust biztosít az adatok szabványos átadására, miután az eszközök között már létrejött a kapcsolat.
- Gondoljunk a **GATT**-ra úgy, mint az adatküldés és fogadás módjára: a kliens eszköz explicit módon lekérheti a szerver adatait, vagy **push üzeneteket** fogad.
- A perifériát **GATT**-szervernek nevezzük, amely a lekérhető attribútumokat (az adatokat), valamint a szolgáltatás- és jellemződefiníciókat tartalmazza.
- A **GATT-kliens** (telefon/PC) az az eszköz, amely kéréseket küld a szervernek. Alapesetben minden tranzakciót a kliens indít

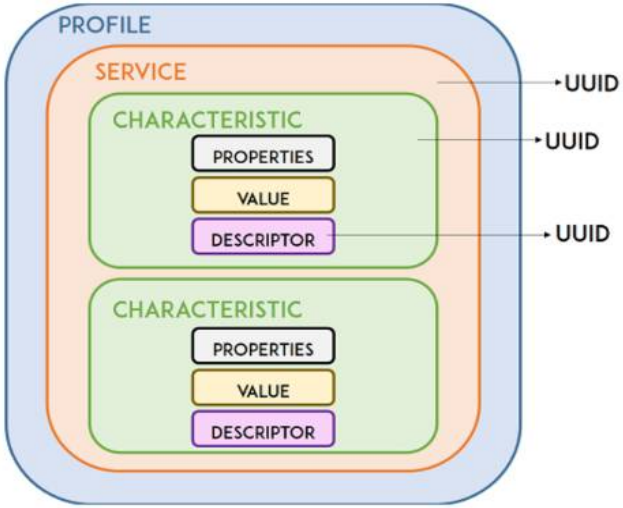


BLE: Tranzakciók és üzenetváltások

- A szerver és a kliens közötti kapcsolat az alábbiakban látható:
- A kliens úgy küld adatot a szervernek, hogy adatokat ír rá: a **Write Request** (visszajelzéssel) vagy a **Write Command** (visszajelzés nélkül) használható.
- A szerver úgy küld adatokat a kliensnek, hogy *jelzést* (**Indication**) vagy *értesítést* (**Notification**) küld a kliensnek. Az egyetlen különbség a kettő között az, hogy megerősítés (nyugtázás) csak az **Indication** használatakor történik
- A kliens bármikor lekérhet adatokat a szerverről egy **Read Request** (olvasási kérés) küldésével.



Szolgáltatások és jellemzők



- A **GATT** tranzakciók az egymásba ágyazott profil, szolgáltatás és jellemző elemeken alapulnak
- **A profil** a szolgáltatások előre meghatározott gyűjteménye. A *Heart Rate Profile* például a *Pulzusszám* és *Eszközinformáció* szolgáltatásból áll (ld: A hivatalosan elfogadott GATT profilok listája)
- **A szolgáltatások** egy vagy több adattömböt, úgynevezett jellemzőket tartalmaznak és mindegyik szolgáltatás egy egyedi numerikus azonosítóval, az úgynevezett UUID-vel különbözteti meg magát a többi szolgáltatástól, amely lehet 16 bites (a hivatalosan elfogadott BLE szolgáltatások esetén) vagy 128 bites (egyedi szolgáltatások esetén)
- **A jellemző** egyetlen adatelemet (attribútum) foglal magában és egy 16 vagy 128 bites UUID-vel különbözteti meg magát a többi *jellemzőtől*
  A **Bluetooth SIG** által meghatározott szabványos jellemzők biztosítják az együttműködést a különféle **BLE**-kompatibilis eszközök között
  - A ***jellemzővel*** végezhető műveleteket *(Read/Write/Notify)* a **tulajdonság** szabja meg
  - A **leírók** további információt adnak az adatról, ill. engedélyezik a **Notify** funkciót

UUID (Universally Unique Identifier)

- Minden *szolgáltatásnak, jellemzőnek* és *leírónak* van egy egyedi 128 bites (16 bájt) UUID azonosítója (Universally Unique Identifier), például:
  `4fafc201-1fb5-459e-8fcc-c5c9c331914b`
- A szabványos szolgáltatások azonban 16 bites rövidített UUID-t is használhatnak, amelyeket a Bluetooth SIG **Assigned Numbers** dokumentuma ismertet

- Amennyiben az alkalmazásunknak saját UUID-re van szüksége, előállíthatjuk ezt az **UUID-generátor webhely** használatával

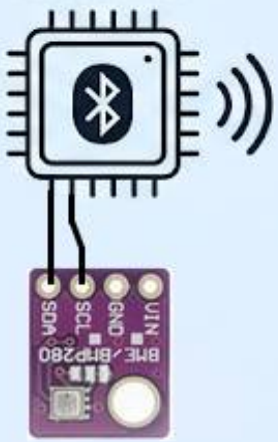
2. projekt: intelligens távadó

- Ebben a projektben egy **ESP32-C3** mikrokontroller és egy **BME280** precíziós szenzor párosával egy **GATT** szervert építünk, ahol a hőmérséklet, páratartalom és légnyomás adatok szabványos, 16 bites azonosítókkal ellátott jellemzőkként (*Characteristic*) jelennek meg
- Az adatfrissítést a leghatékonyabb **Notify** metódussal oldjuk meg: a szenzor nem vár a kliens kérdésére, hanem eseményvezérelt módon, azonnal küldi a friss mért értékeket a telefonunkra vagy a monitorozó alkalmazásunkra
- **Szabványkövetés:** Az **0x181A** (*Environmental Sensing*) szolgáltatást használjuk, így bármilyen szabványos BLE app azonnal felismeri az adatokat.
- **Hatékonyság:** A **Notify** használata biztosítja a legalacsonyabb fogyasztást a kliens oldalon, hiszen nincs szükség felesleges lekérdezésekre (*Polling*).
- **Adatintegritás:** A szabványos UUID-k használata biztosítja, hogy a szoftverünk ne csak "valamilyen számokat", hanem egzakt fizikai mennyiségeket közvetítsen

BLE GATT STRUKTÚRA: BME280 KÖRNYEZETI SZENZOR

ESP32-C3 PERIPHERAL

ESP32-C3



BME280

SERVICE: KÖRNYEZETI ADATOK (Environmental Sensing)

UUID: **181A**

CHARACTERISTICS: ADATPONTOK

HŐMÉRSÉKLET
(TEMP_CHAR_UUID)
UUID: **2A6E**
Properties: [Read] [Notify]

PÁRATARTALOM
(HUM_CHAR_UUID)
UUID: **2A6F**
Properties: [Read] [Notify]

NYOMÁS
(PRES_CHAR_UUID)
UUID: **2A6D**
Properties: [Read] [Notify]

KLIENSEK: DOKUMENTÁCIÓ ÉS PREZENTÁCIÓ

NRF CONNECT MOBILE

PYTHON (BLEAK)

```
async with BleakClient(address) as client:
  ...
  data = await client.read_gatt_char(TEMP_CHAR_UUID)
  ...
```

Feliratkozás és Értesítés

```
// 1. Karakterisztika létrehozása (READ + NOTIFY)
pService->createCharacteristic(
    TEMPERATURE_CHAR_UUID,
    NIMBLE_PROPERTY::READ | NIMBLE_PROPERTY::NOTIFY
);

// 2. Szenzoradat fixpontossá alakítása (századfokok)
float temperature = bme.readTemperature(); // pl. 24.56f
int16_t t_val = (int16_t)(temperature * 100); // 2456 (századfok)


// 3. Adat frissítése és Notify küldése
// Szabványos, tiszta 2-bájtos adatáttvitel
 pTempChar->setValue((uint8_t*)&t_val, 2);
 pTempChar->notify();
```

(#define TEMPERATURE_CHAR_UUID "2A6E")

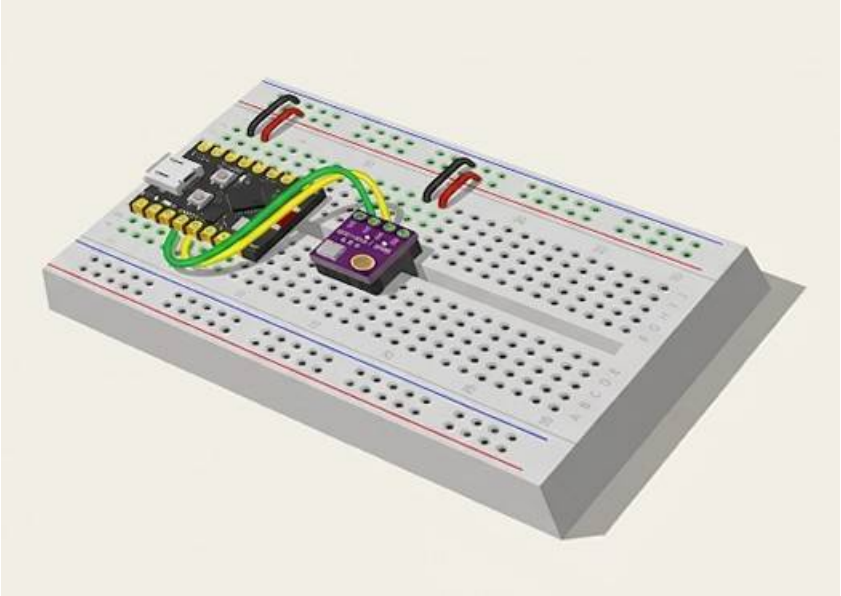
A jellemző olvasható, és értesítés kérhető

A fentiekhez hasonlóan frissítjük a pHumChar és pPresChar mutatókkal címzett jellemzőket is

ble_notify.ino – 3/1.

- Ez a program egy GATT-szerver megvalósítása ESP32-C3 platformon, amely a BME280 szenzor adatait (hőmérséklet, páratartalom, légnyomás) közvetíti Bluetooth LE kapcsolaton. A rendszer az iparági standardnak megfelelő 16 bites UUID azonosítókat és hatékony Notify-alapú aszinkron adatközlést alkalmaz

```
#include <Wire.h>
#include <WiFi.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>
#include <NimBLEDevice.h>
#define I2C_SDA 8
#define I2C_SCL 9
#define SERVICE_UUID        "181A"
#define TEMP_CHAR_UUID      "2A6E"
#define HUM_CHAR_UUID       "2A6F"
#define PRES_CHAR_UUID      "2A6D"

Adafruit_BME280 bme;
NimBLEServer *pServer;
NimBLECharacteristic *pTempChar;
NimBLECharacteristic *pHumChar;
NimBLECharacteristic *pPresChar;
```



ble_notify.ino – 3/2.

```
void setup() {
  Serial.begin(115200);
  WiFi.mode(WIFI_OFF);
  Wire.begin(I2C_SDA, I2C_SCL);
  if (!bme.begin(0x76, &Wire)) { Serial.println("BME280 hiba!"); while (1);  }
  NimBLEDevice::init("C3-BME280");
  pServer = NimBLEDevice::createServer();
  NimBLEService *pService = pServer->createService(SERVICE_UUID);
  pTempChar = pService->createCharacteristic(TEMP_CHAR_UUID, NIMBLE_PROPERTY::READ | NIMBLE_PROPERTY::NOTIFY);
  pHumChar  = pService->createCharacteristic(HUM_CHAR_UUID,  NIMBLE_PROPERTY::READ | NIMBLE_PROPERTY::NOTIFY);
  pPresChar = pService->createCharacteristic(PRES_CHAR_UUID, NIMBLE_PROPERTY::READ | NIMBLE_PROPERTY::NOTIFY);
  Pservice->start();

  NimBLEAdvertising *pAdvertising = NimBLEDevice::getAdvertising();
  NimBLEAdvertisementData advData;    // Hirdetési csomag (Service UUID + Flags)
  advData.setFlags(BLE_HS_ADV_F_DISC_GEN | BLE_HS_ADV_F_BREDR_UNSUP);
  advData.addServiceUUID(NimBLEUUID(SERVICE_UUID));
  pAdvertising->setAdvertisementData(advData);
  NimBLEAdvertisementData scanRes;    // Válasz csomag (Név) - Ez váltja ki a "Scan Response"-t
  scanRes.setName("C3-BME280");
  pAdvertising->setScanResponseData(scanRes);
  pAdvertising->start();              // Hirdetés indítás
}
```

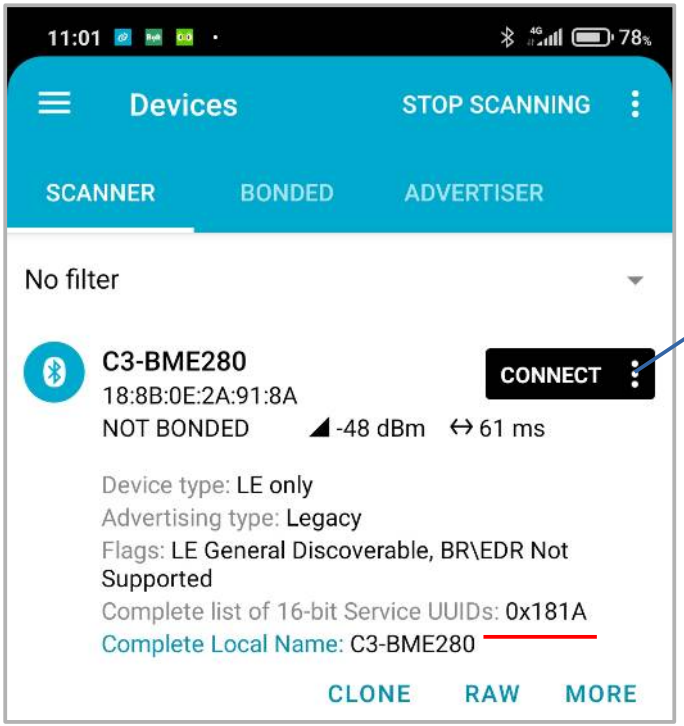
ble_notify.ino – 3/3.

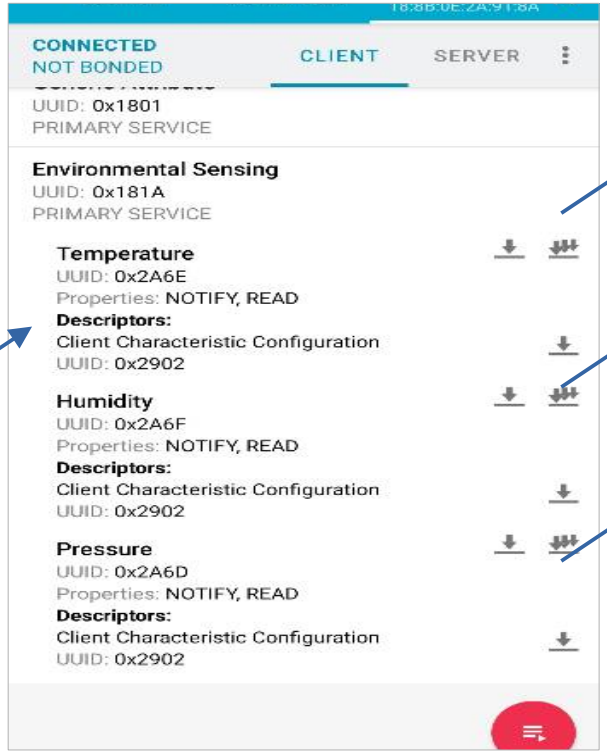
```
void loop() {
  static unsigned long lastMillis = 0;
  if (millis() - lastMillis >= 3000) {
    lastMillis = millis();
    float t = bme.readTemperature();
    float h = bme.readHumidity();
    float p_abs = bme.readPressure() / 100.0F;
    float p_sea = bme.seaLevelForAltitude(139.0, p_abs); // Tengerszintre átszámított légnyomás
    int16_t t_val = (int16_t)(t * 100);
    int16_t h_val = (int16_t)(h * 100);
    uint32_t p_val = (uint32_t)(p_sea * 10);   // 1000-rel kellene szorozni a Pa egység miatt
    pTempChar->setValue((uint8_t*)&t_val, 2);
    pHumChar->setValue((uint8_t*)&h_val, 2);
    pPresChar->setValue((uint8_t*)&p_val, 4);
    if (pServer->getConnectedCount() > 0) {
      PtempChar->notify(); pHumChar->notify(); pPresChar->notify();
    }  else {
      if (!NimBLEDevice::getAdvertising()->isAdvertising()) {  // Ha nincs kapcsolat és nem hirdetünk!
        NimBLEDevice::startAdvertising();      //Hirdetés kézi újraindítása
      }
    }
  }
}
```

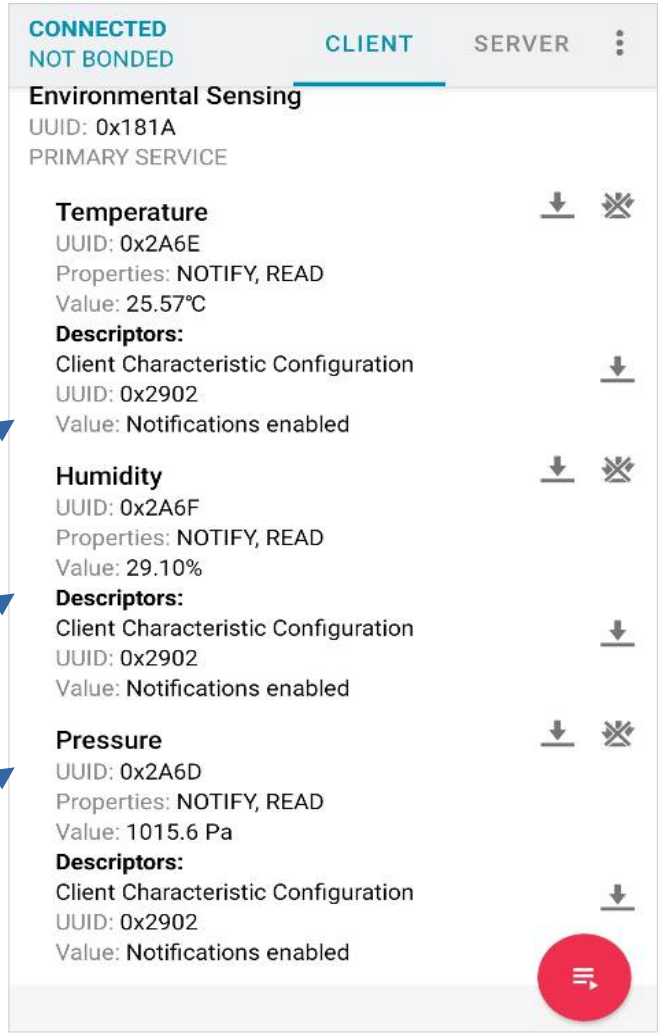
Itt egy kicsit eltértünk a szabványtól, mert 0.1 Pa helyett 0.1 hPa egységben küldjük ki a légnyomás adatot

A Notify frissítések ellenőrzése

Az **ESP32-C3** üzeneteit az [nRF Connect for Mobile] alkalmazással ellenőriztük. Az értesítéseket (Notify) a „háromnyilas” gombokkal engedélyezhetjük







ble_dashboard.py

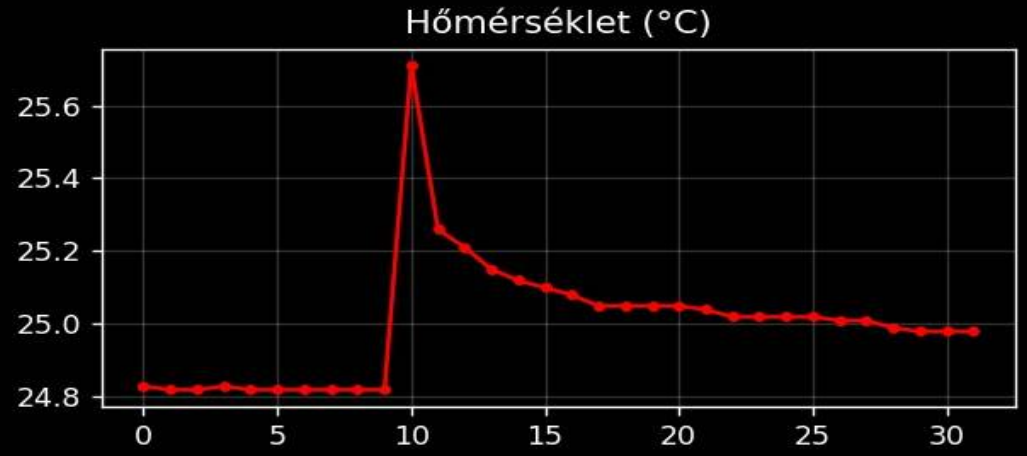
- A Python „műszerfal” program két párhuzamosan futó folyamatra épül, amelyek egy közös adatterületen keresztül kommunikálnak egymással
- **Komunikációs folyamat** - ez a modul felelős a külvilággal való kapcsolatért
  - **Keresés és Csatlakozás:** megkeresi a **C3-BME280** nevű eszközt és csatlakozik
  - **Feliratkozás:** kéri az eszköztől a szenzoradatok automatikus küldését (**Notify**)
  - **Aszinkron adatfogadás:** ha új adat érkezik, Amint új csomag érkezik, beazonosítja (UUID alapján), dekódolja a bináris adatot, és elmenti a memóriába
  - **Öngyógyító ciklus:** Ha a kapcsolat megszakad, azonnal újraindítja a keresést
- **Megjelenítési folyamat** - Ez a modul felelős a felhasználói élményért
  - **Időzített frissítés:** Másodpercenként egyszer mintát vesz a memóriából
  - **Grafikus leképezés:** A memóriában tárolt utolsó 60 adatpontból dinamikus grafikonokat rajzol (hőmérséklet, páratartalom, légnyomás).
  - **Digitális kijelzés:** Az aktuális értékeket "műszerfal" jelleggel jeleníti meg

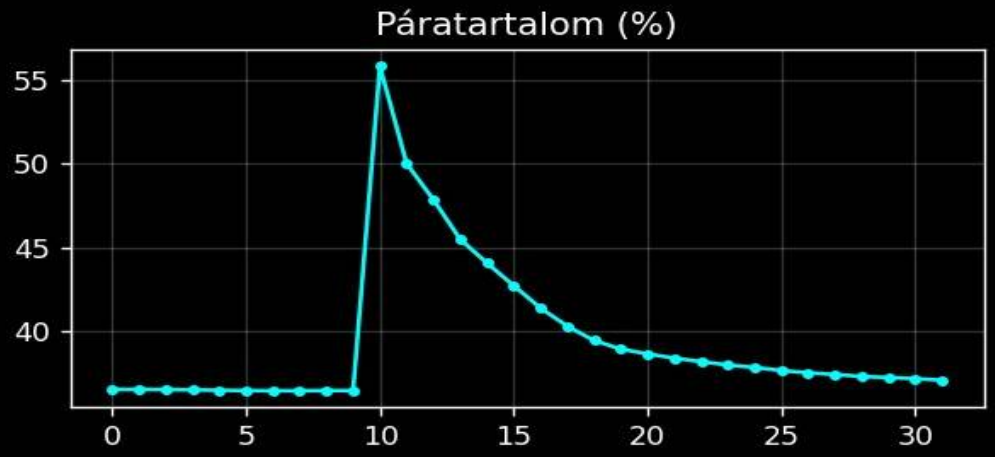
BME280 & ESP32-C3 BLE Dashboard

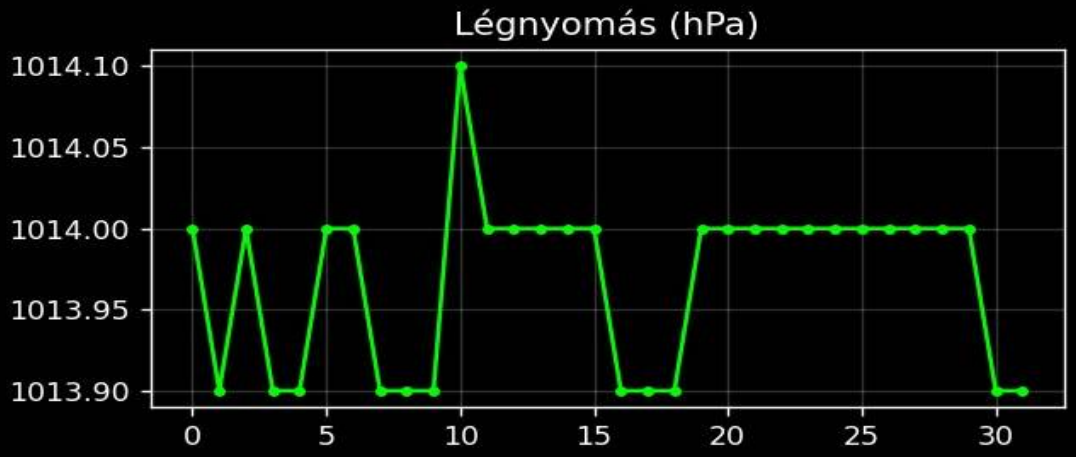
HŐMÉRSÉKLET: 24.98 °C

PÁRATARTALOM: 37.08 %

LÉGNYOMÁS: 1013.9 hPa







Az ESP32 C3 Super Mini kártya kivezetései

Megjegyzés: Az A5 analóg bemenet (ADC2) nem használható, ha a WiFi használatban van!