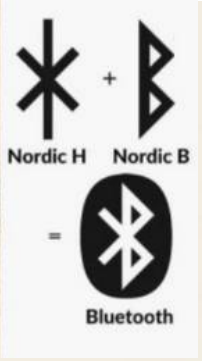


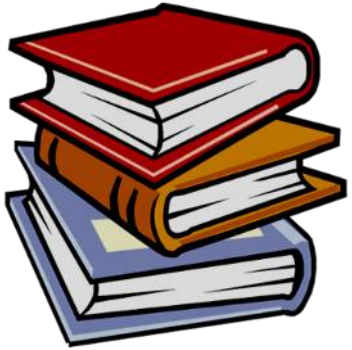

# Bluetooth Low Energy & ESP32-C3 – II. rész




Bluetooth®

# Felhasznált és ajánlott irodalom

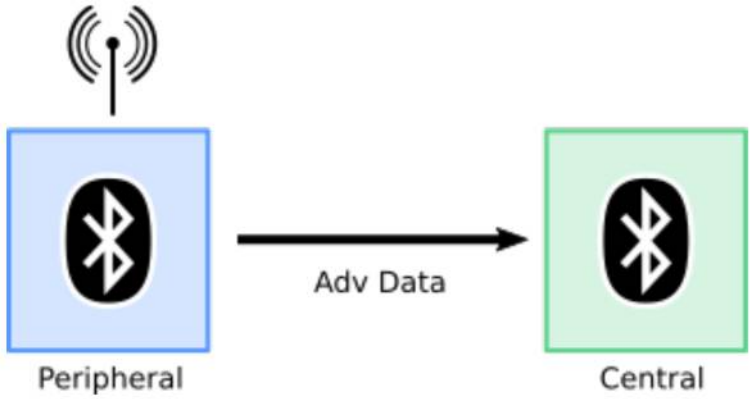
- **Leírások, dokumentáció**
  - Adafruit: Introduction to Bluetooth Low Energy
  - Timothy Woo: ESP32 + Android + Arduino = Awesome
  - Neil Kolban: Kolban's book on ESP32
  - Neil Kolban: BLE C++ Guide.pdf
  - ESPRESSIF: ESP32 Arduino Core Documentation



- **Szoftver segédlet**
  - Version 4 UUID Generator
  - Nordic Semiconductor ASA: nRF Connect for Mobile
  - Ryan Powell: NimBLE-Arduino library
  - MIT Center for Constructive Computing: MIT App Inventor
  - Thunkable Inc.: Thunkable - No-Code Mobile App Development Platform
  - Adafruit: BME280 sensor library

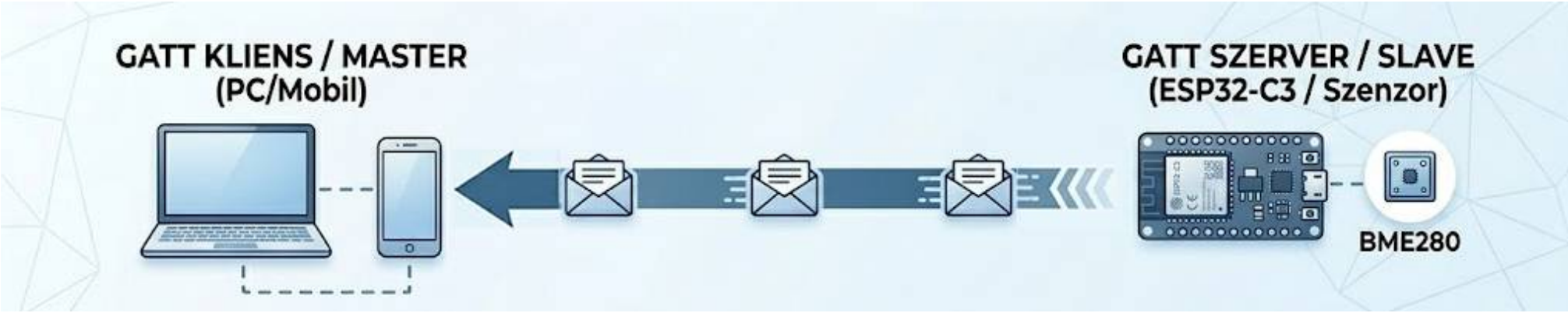
# BLE: Hogyan találnak egymásra az eszközök?

- **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



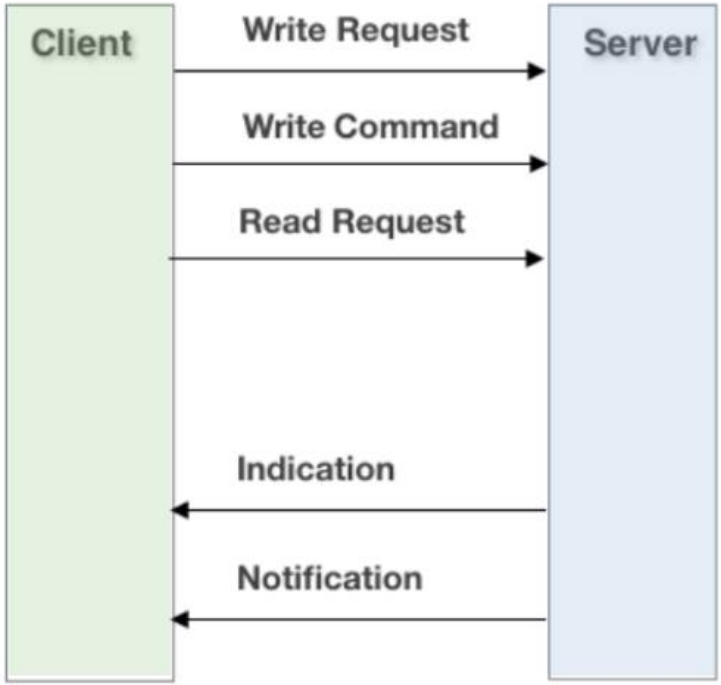
# BLE: Hogyan zajlik az adatcsere? (GATT)

- 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

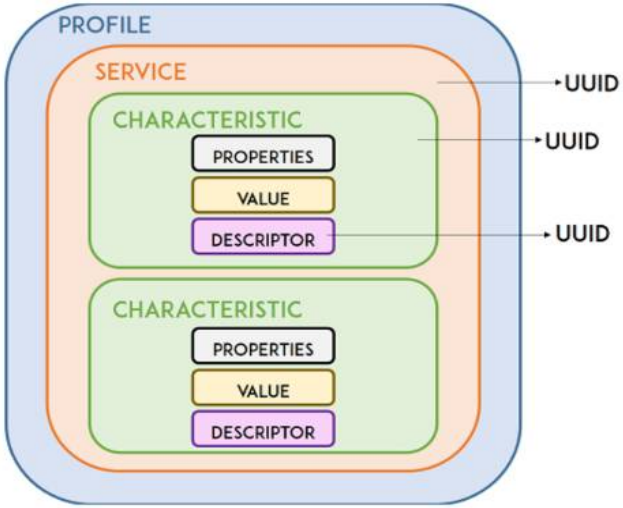


# BLE: Tranzakciók és üzenetváltások

- 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)

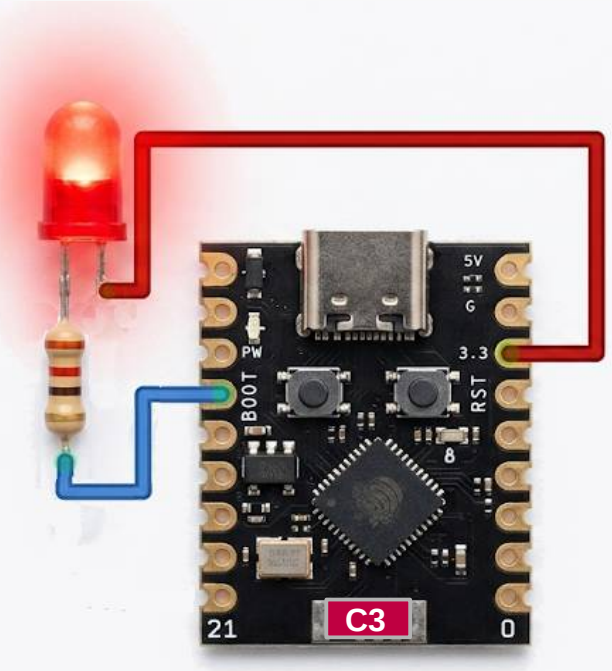
- Minden *szolgáltatásnak, jellemzőnek* és *leírónak* van egy egyedi 128 bites (16 bájt) UUID azonosítója (Universally Unique Identifier), például:
  `4fafc201-1fb5-459e-8fcc-c5c9c331914b`
- A szabványos szolgáltatások azonban 16 bites rövidített UUID-t is használhatnak, amelyeket a Bluetooth SIG **Assigned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


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

# 1. projekt: ESP32_BLE_Demo

**ESP32 BLE + Android + Arduino IDE = AWESOME** címmel egy érdekes projektet mutatott be Timothy Woo , ezt most **ESP32-C3**-mal és a **NimBLE** könyvtárral építjük meg. A szerver program legfőbb tanulsága a **Read, Write, Notify** eseményvezérelt kezelése

A program a **NUS** (Nordic UART Service) protokollt használja, ami egy olyan egyedi BLE szolgáltatás, amely a hagyományos soros porti (UART) kommunikációt emulálja vezeték nélkül, lehetővé téve a kétirányú szöveges üzenetváltást a kliens és a szerver között




# Eseményvezérelt kiszolgálás

- A **BLE** kommunikáció nem folyamatos adatfolyam, hanem diszkrét események sorozata. A szerver (ESP32) passzív állapotban vár, amíg a kliens (pl. mobil) interakciót nem kezdeményez. Ekkor a rendszer automatikusan meghívja a megfelelő visszahívási függvényt (Callback)
- **Kezelendő eseménytípusok:**
  - **READ** (Olvasás) – amikor a mobilalkalmazás lekéri egy jellemző aktuális állapotát
    Szerver feladata: Az esemény pillanatában frissíteni a karakterisztika értékét, biztosítva, hogy a kliens ne elavult, hanem "Just-in-Time" generált adatot kapjon.
  - **WRITE** (Írás) – a mobilalkalmazás adatot küld az eszköznek (pl. LED kapcsolás)
    Szerver feladata: A beérkező adatcsomag fogadása, értelmezése és a fizikai beavatkozás (GPIO írás) azonnali végrehajtása
  - **CONNECTIVITY** (Kapcsolatkezelés) – amikor kliens csatlakozik, vagy lecsatlakozik
    Szerver feladata: A kapcsolat állapotának adminisztrálása, és szétkapcsolás esetén a hirdetés (Advertising) automatikus újraindítása a folyamatos elérhetőség érdekében

# ESP32-C3 NImBLE CALLBACKS

```
class MyTX: public NimBLECharacteristicCallbacks {
    void onRead(NimBLECharacteristic* pC, NimBLEConnInfo& info) override {
        char buf[8];
        dtostrf(lastMeas, 1, 1, buf);
        pC->setValue(buf);
    }
};

class MyRX: public NimBLECharacteristicCallbacks {
    void onWrite(NimBLECharacteristic* pC, NimBLEConnInfo& info) override {
        std::string val = pC->getValue();
        digitalWrite(8, (val.find("A") != -1) ? LOW : HIGH);
    }
};

class MyServer: public NimBLEServerCallbacks {
    void onConnect(NimBLEServer* pS, NimBLEConnInfo& i) override { connected = true; }
    void onDisconnect(NimBLEServer* pS, NimBLEConnInfo& i, int r) override {
        connected = false;
        NimBLEDevice::startAdvertising();
    }
};
```

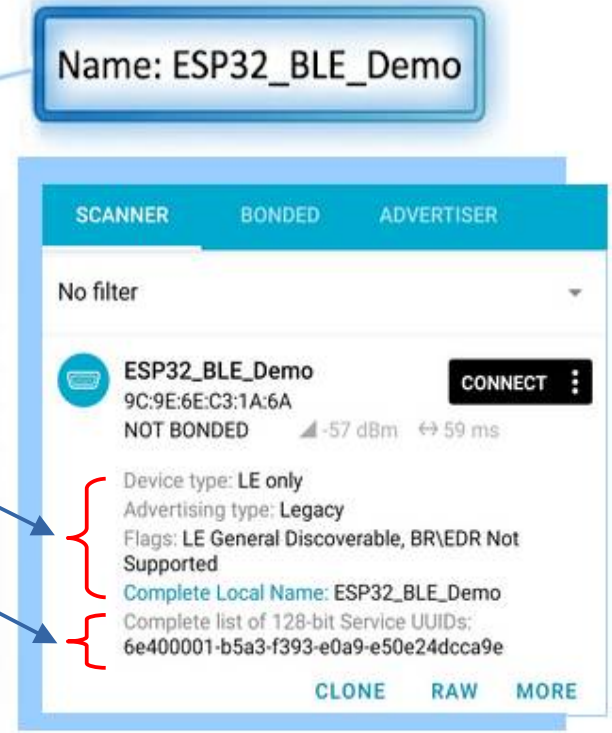
Connect

Disconnect

# A Reklám- és Scan-Response Csomag Összeálllítása (ESP32-C3 / C++)

```
NimBLEAdvertising* pAdv = NimBLEDevice::getAdvertising();

NimBLEAdvertisementData advData;
advData.setFlags(0x06);
advData.setName("ESP32_BLE_Demo");
pAdv->setAdvertisementData(advData);

NimBLEAdvertisementData scanData;
scanData.addServiceUUID(NimBLEUUID(S_UUID));
pAdv->setScanResponseData(scanData);

pAdv->enableScanResponse(true);
pAdv->start();
```

A 31 bites korlát miatt a név és a service UUID nem fér bele együtt az elsődleges hirdetési csomagba



# ESP32_BLE_Demo
## 2/1. oldal

```
#include "NimBLEDevice.h"
#define S_UUID "6E400001-B5A3-F393-E0A9-E50E24DCCA9E"
#define RX_UUID "6E400002-B5A3-F393-E0A9-E50E24DCCA9E"
#define TX_UUID "6E400003-B5A3-F393-E0A9-E50E24DCCA9E"
NimBLECharacteristic* pTX;
bool connected = false; float lastMeas = 0.0;

class MyTX: public NimBLECharacteristicCallbacks {
    void onRead(NimBLECharacteristic* pC, NimBLEConnInfo& info) override {
        char buf[8]; dtostrf(lastMeas, 1, 1, buf);
        pC->setValue(buf);
    }
};

class MyRX: public NimBLECharacteristicCallbacks {
    void onWrite(NimBLECharacteristic* pC, NimBLEConnInfo& info) override {
        std::string val = pC->getValue();
        digitalWrite(8, (val.find("A") != -1) ? LOW : HIGH);
    }
};

class MyServer: public NimBLEServerCallbacks {
    void onConnect(NimBLEServer* pS, NimBLEConnInfo& i) override { connected = true; }
    void onDisconnect(NimBLEServer* pS, NimBLEConnInfo& i, int r) override {
        connected = false; NimBLEDevice::startAdvertising();
    }
};
```

**Logikai blueprint:** Az alapértelmezett callback-ek felülírása saját rutinokkal.

**Példányosítás később (a következő oldalon):** Ez csak a "szabályrendszer", a funkciók aktiválása a következő lépésben történik.

# ESP32_BLE_Demo
## 2/2. oldal

```
void setup() {
    pinMode(8, OUTPUT); digitalWrite(8, HIGH);
    NimBLEDevice::init("ESP32_BLE_Demo");
    NimBLEServer* pS = NimBLEDevice::createServer();
    pS->setCallbacks(new MyServer());
    NimBLEService* pService = pS->createService(S_UUID);
    pService->createCharacteristic(RX_UUID, NIMBLE_PROPERTY::WRITE)->setCallbacks(new MyRX());
    pTX = pService->createCharacteristic(TX_UUID, NIMBLE_PROPERTY::READ | NIMBLE_PROPERTY::NOTIFY);
    pTX->setCallbacks(new MyTX());
    pService->start();
    NimBLEAdvertising* pAdv = NimBLEDevice::getAdvertising(); NimBLEAdvertisementData advData;
    advData.setFlags(0x06); advData.setName("ESP32_BLE_Demo");
    pAdv->setAdvertisementData(advData);
    NimBLEAdvertisementData scanData;
    scanData.addServiceUUID(NimBLEUUID(S_UUID));
    pAdv→setScanResponseData(scanData); pAdv->enableScanResponse(true); pAdv->start();
}

void loop() {
    lastMeas = 25.4;
    if (connected) {
        char buf[8]; dtostrf(lastMeas, 1, 1, buf);
        pTX->setValue(buf); pTX->notify();
    }
    delay(2000);
}
```

Server

Service

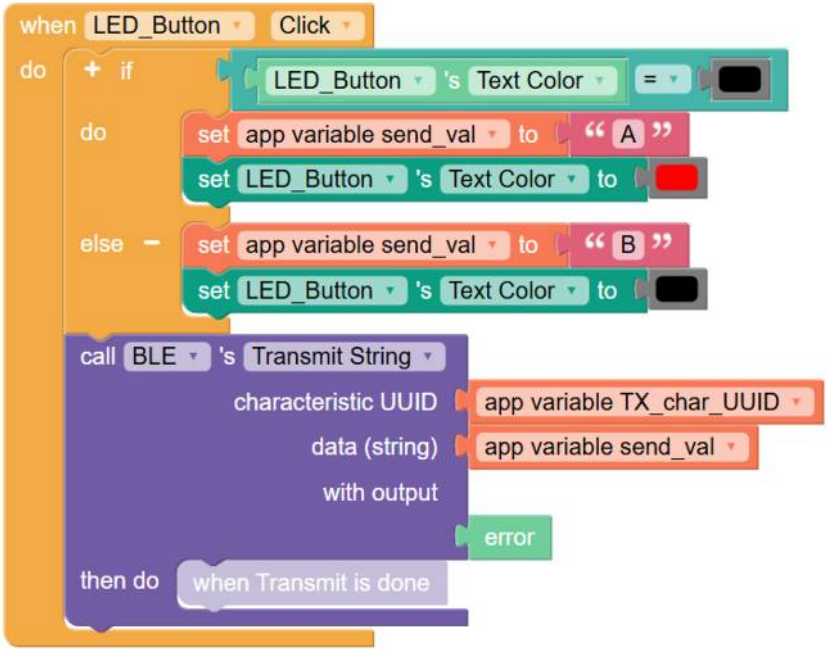
Advertising

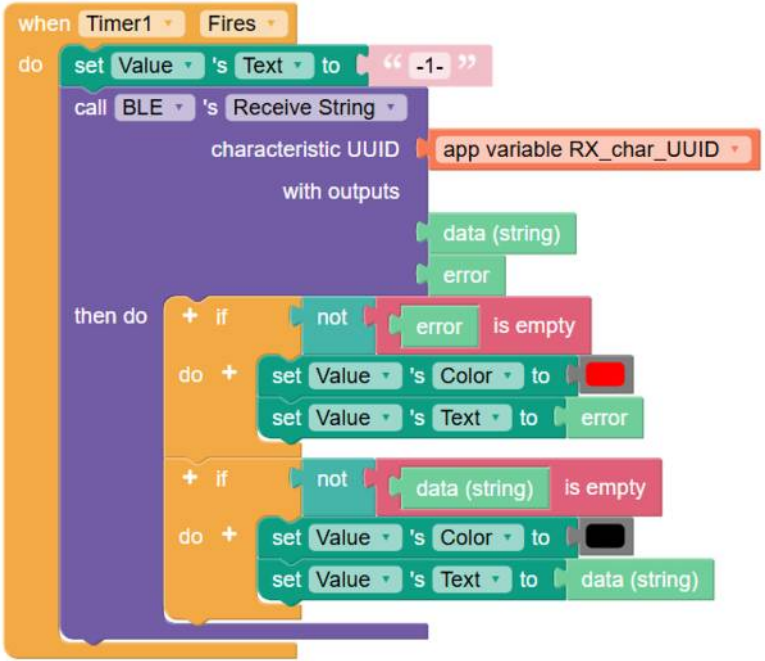
A hardver inicializálások után létrehozzuk a szervert, a szolgáltatást és a karakterisztikákat, példányosítjuk a visszahívási függvényeket, majd összeállítjuk hirdetési csomagot

A **loop függvényben** mérési adatot szimulálunk, s itt szolgáljuk ki a **Notify** igényeket is

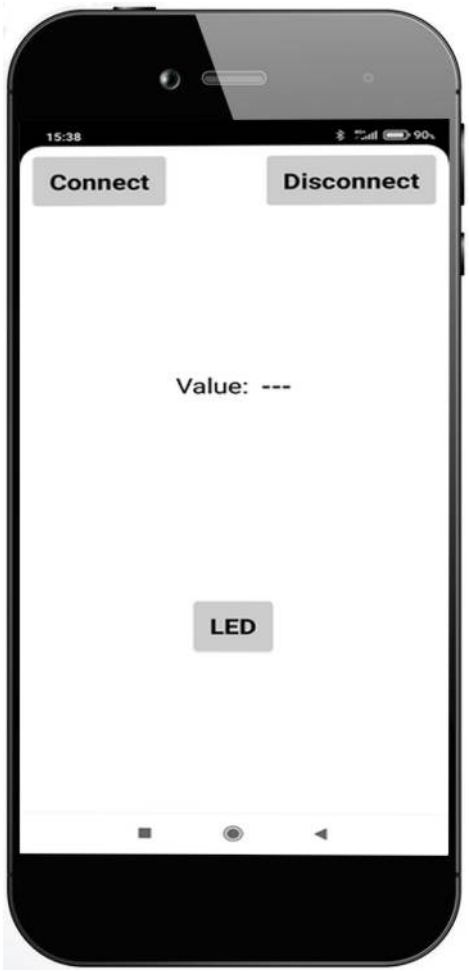
# Az ESP32_BLE_Demo app (részlet)


Az eredet alkalmazást Timothy Woo frissítette, hogy a [Thunkable](Thunkable) újabb kiadásával kompatibilis legyen. Az ábrán a LED kezelést és az adatok fogadását mutatjuk be



# A BLE_Demo app futási eredménye

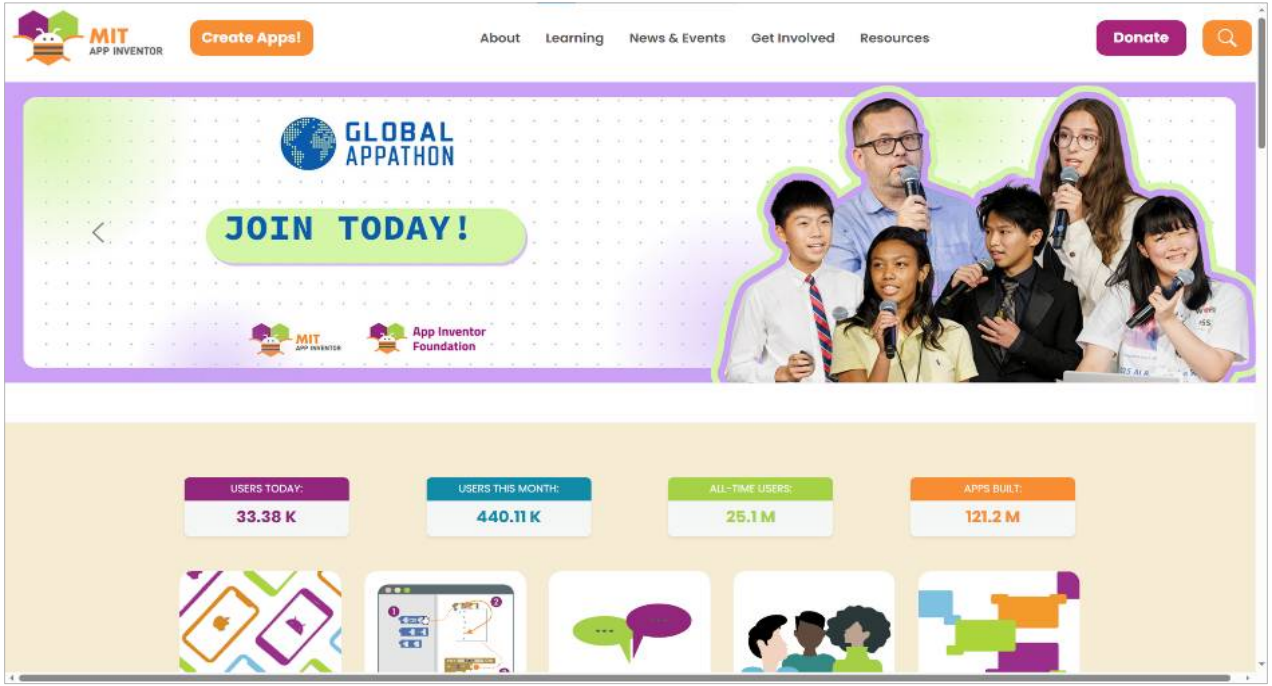




# 2. projekt: Android BLE kliens készítése

# MIT App Inventor

- A MIT App Inventor ingyenes, korlátozásmentes vizuális fejlesztőkörnyezet, amellyel bárki – akár gyerekek is – készíthet mobilappot, és amely a hagyományos programozásnál gyorsabban teszi lehetővé összetett alkalmazások létrehozását

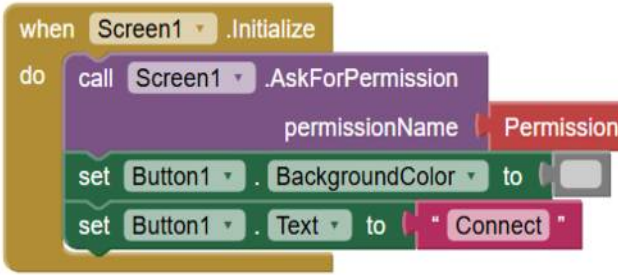



# BME280_Notify

- Az Android alkalmazás egy BLE (Bluetooth Low Energy) Kliens ami az előző előadásban bemutatott **ESP32-C3 + BME280** környezeti távadó adatait fogadja és jeleníti meg
- **MAC cím alapú direkt elérés:** Az alkalmazás előre definiált MAC cím alapján keresi meg a szervert kezdeményez kapcsolatot
- **GATT Profil alapú kommunikáció:** Az alkalmazás az ESP32-C3 által hirdetett **0x181A** (Environmental Sensing) szabványos szolgáltatáshoz csatlakozik. A kliensoldali logika a konkrét karakterisztika UUID-k (**2A6E, 2A6F, 2A6D**) alapján azonosítja a beérkező adatfolyamokat
- **Aszinkron adatfeldolgozás** (Notification): Az alkalmazás nem lekérdezéses módszerrel dolgozik, hanem a Notify metódusra iratkozik fel. Így az Android eszköz csak akkor frissíti a UI-t és végez számításokat, ha a szerver oldalon tényleges változás történt
- **Adatkonverziós logika:** Mivel a szerver fixpontos egész számokként küldi az adatokat (pl. a hőmérsékletet századfok pontossággal), az App Inventor környezetben helyre kell állítanunk az adatokat a mértékegységek (Celsius, %, hPa) kezeléséhez.

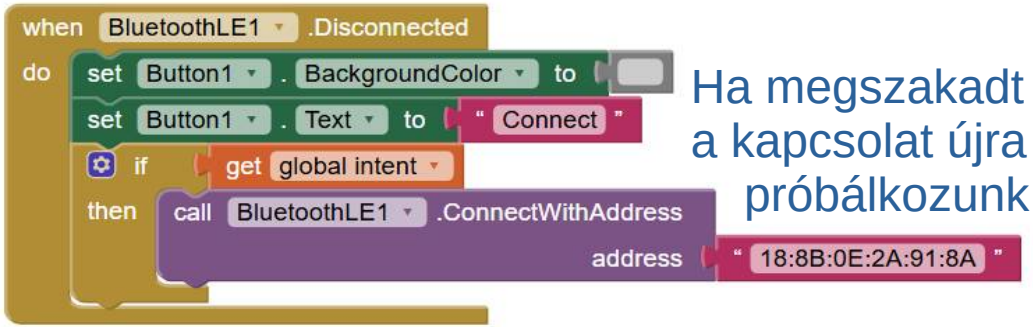
  Az adatfeldolgozás során a hőmérséklet és páratartalom előjeles short integerként, míg a légnyomás 4 elemből álló bájtlistaként érkezik, ezeket az alkalmazás szoftveresen konvertálja a végleges formátumra


Globális változók és konstansok

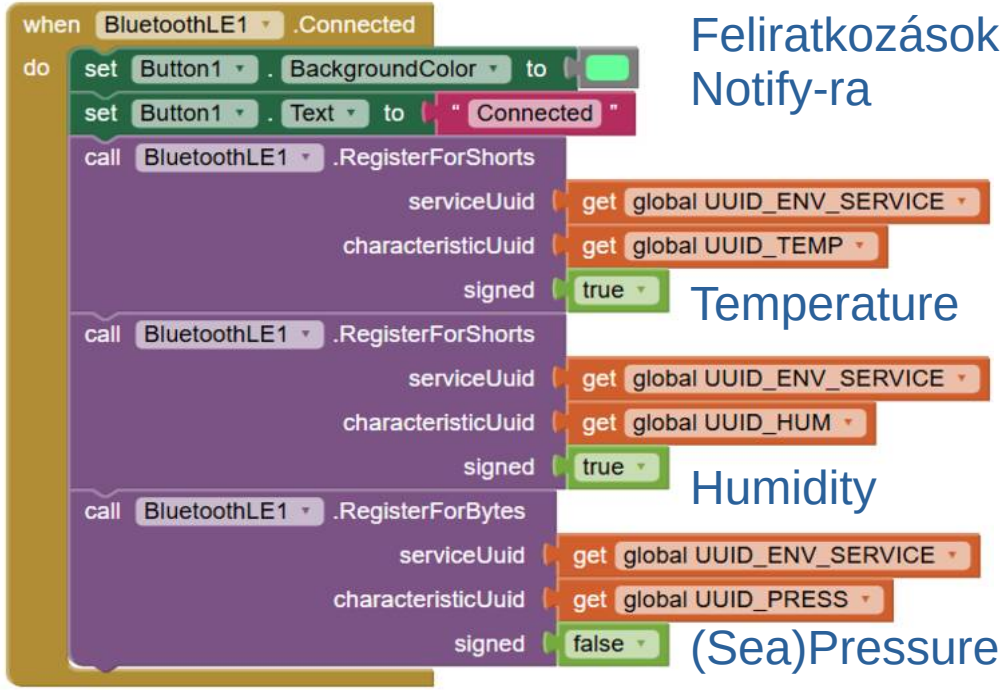



Az alkalmazás engedélyt kér

Kezdetben még nincs kapcsolat




Ha megszakadt a kapcsolat újra próbálkozunk

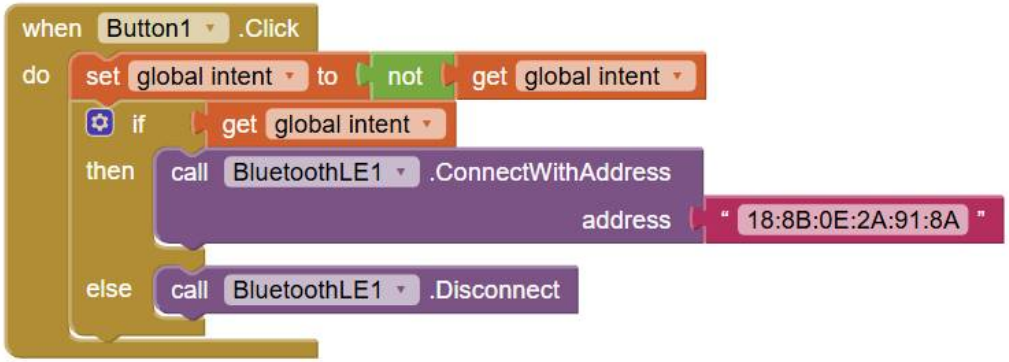



Feliratkozások Notify-ra

Temperature

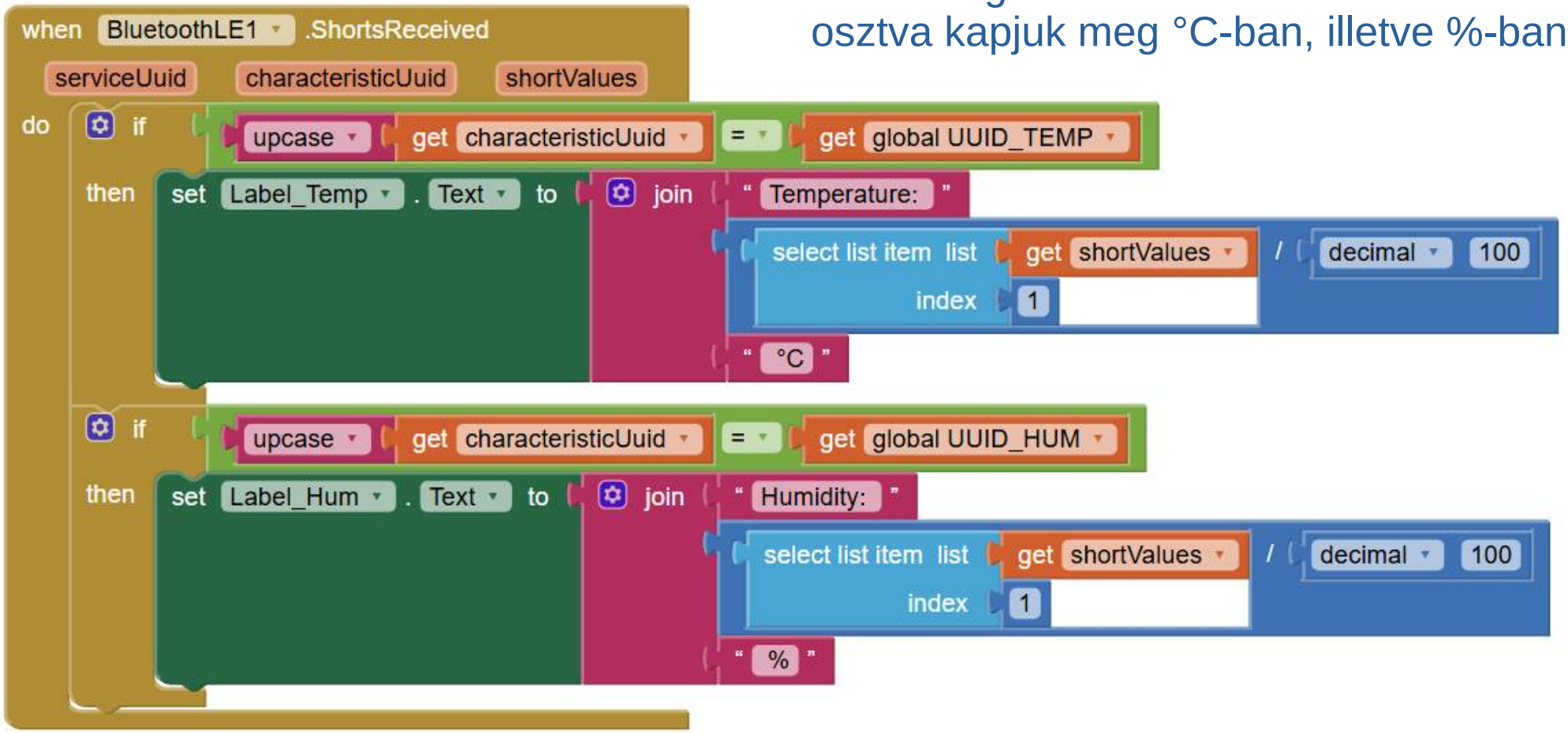
Humidity

(Sea)Pressure



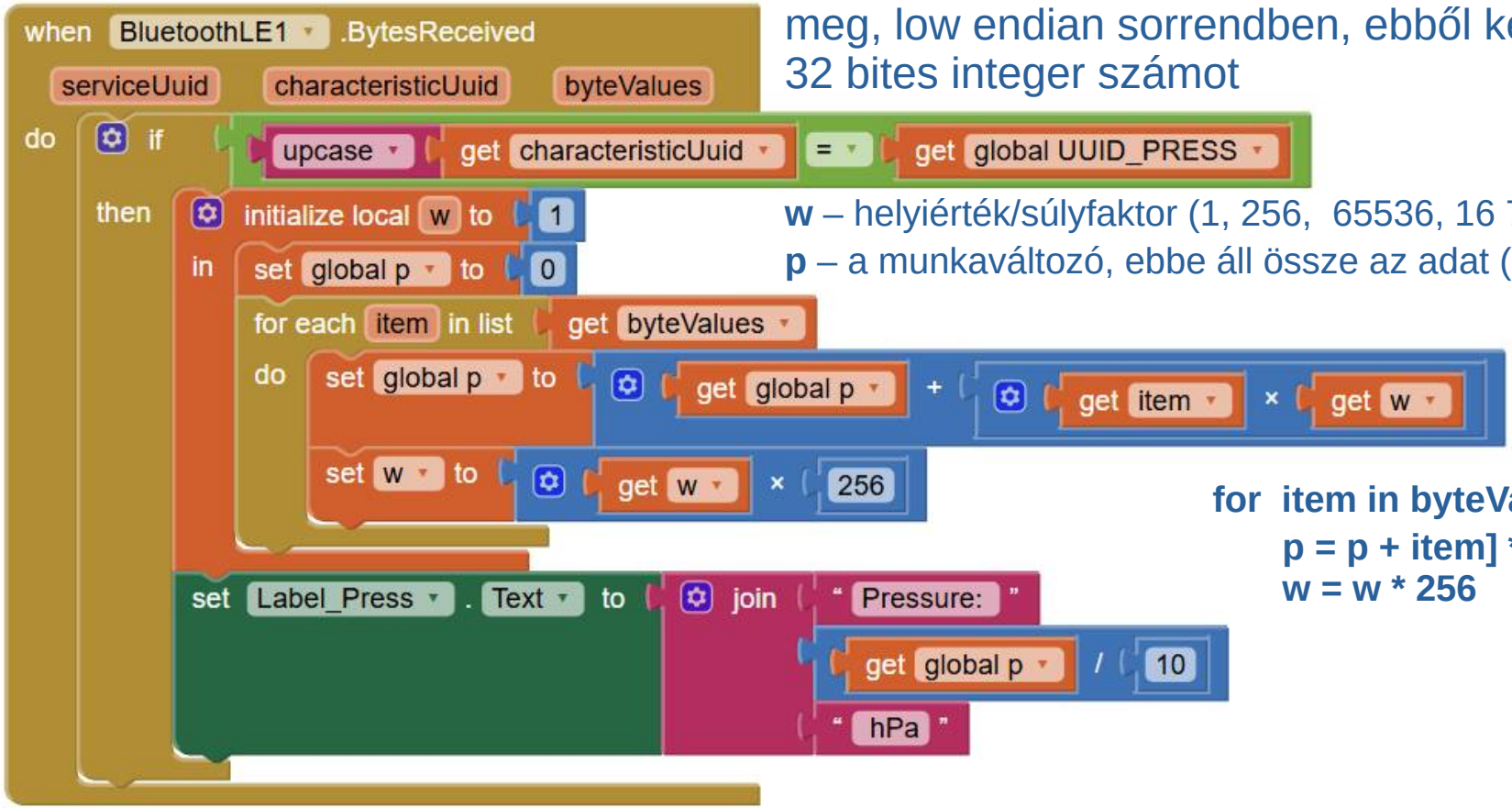
# BME280_Notify – 3/2.

A hőmérséklet és páratartalom előjeles short integerként érkezik és 100-zal osztva kapjuk meg °C-ban, illetve %-ban



# BME280_Notify – 3/3.

A 32 bites légnyomás adatot négy bájtra bontva kapjuk meg, low endian sorrendben, ebből kell helyreállítani a 32 bites integer számot




**w** – helyiérték/súlyfaktor (1, 256, 65536, 16 777 216)

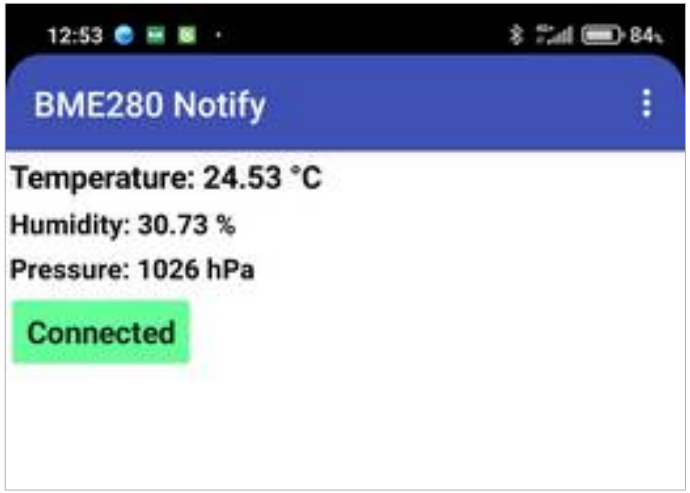
**p** – a munkaváltozó, ebbe áll össze az adat (kezdetben 0)

**for item in byteValues:**
**p = p + item] * w**
**w = w * 256**

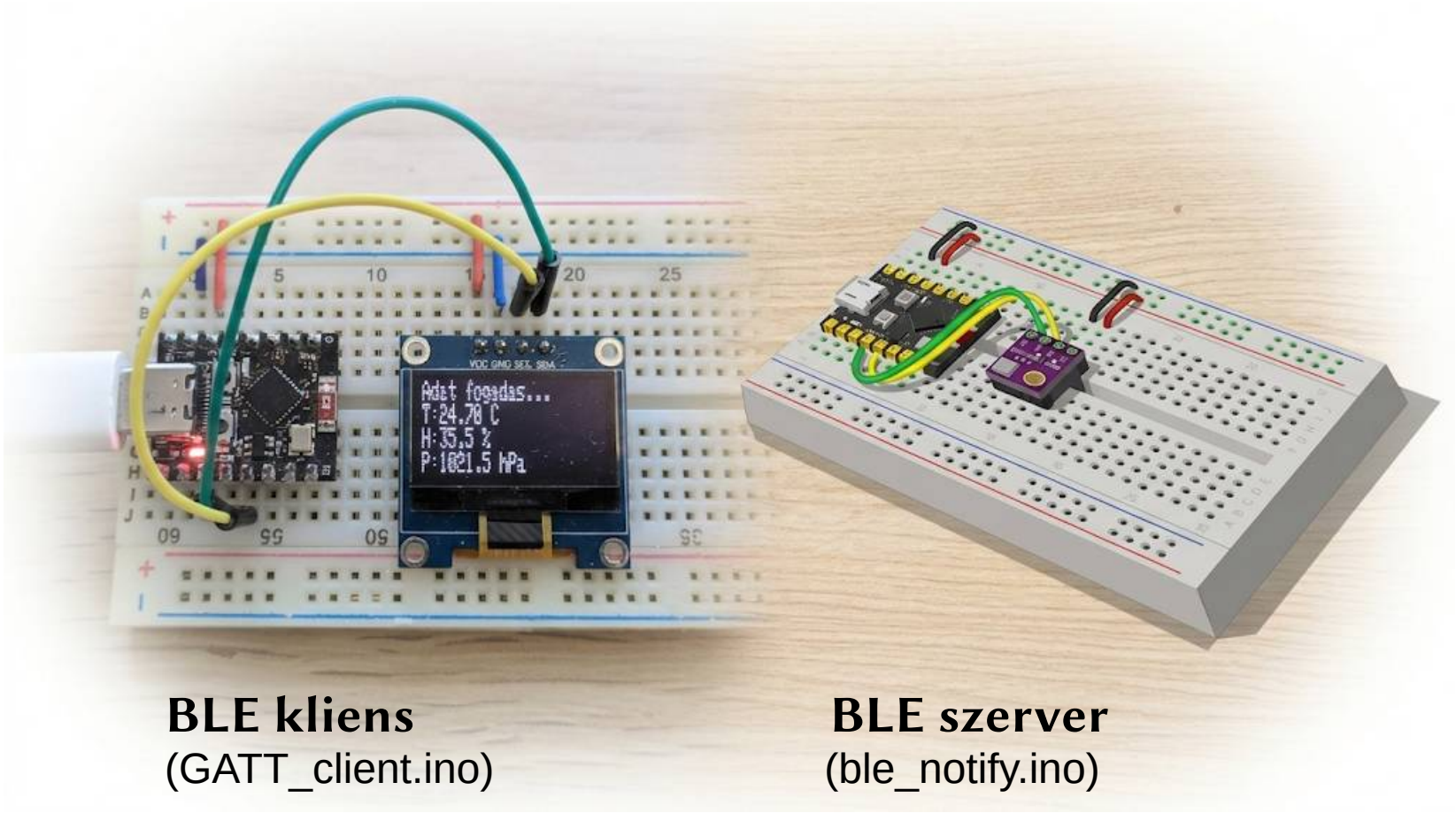
# BME280_Notify

A futási eredmény az alábbi ábrán látható.

**Házi feladat:** cicomázzuk  a megjelenés pl. úgy, ahogy a második kép mutatja. (Csak erős idegzetűeknek!)




# 3. projekt: ESP32-C3 mint GATT kliens




**BLE kliens**
(GATT_client.ino)

**BLE szerver**
(ble_notify.ino)

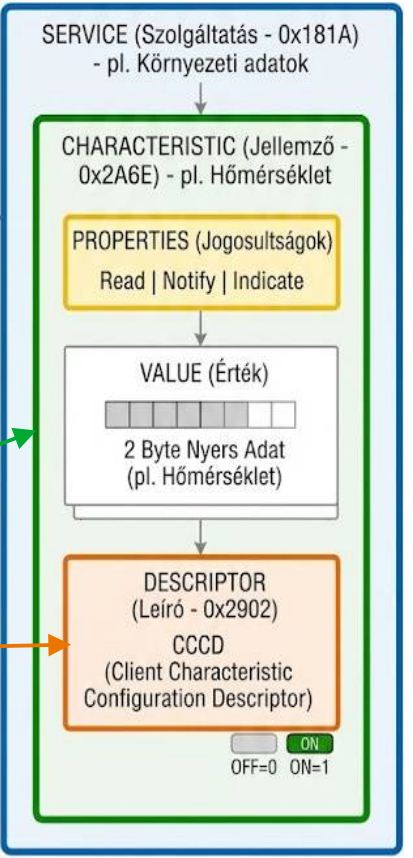
# A GATT Kliens szerepe és az adatkapcsolat folyamata

A **BLE** kommunikáció során a Kliens az aktív fél, felkutatja az önmagát hirdető szervert és vezérli az adatkapcsolatot, illetve az adatfolyamot:

- **Scanning** (Megfigyelés): A kliens figyeli a hirdetési csomagokat (Advertising Packets). Feladata a keresett eszköz azonosítása (Név, UUID, vagy MAC alapján) a "zajban"
- **Connecting** (Kapcsolódás): A célpont kiválasztása után a kliens kezdeményezi a pont-pont kapcsolatot. A szerver ekkor leállítja a hirdetést
- **Service Discovery** (Felfedezés): A kliens lekéri a szerver belső szerkezetét. Megkeresi a specifikus szolgáltatásokat (pl. 0x181A) és a hozzájuk tartozó jellemzőket
- **Adatkezelési módok:**
  - **Poll** (Lekérdezés): A kliens eseti jelleggel adatot kér. Erőforrás-igényes és lassú
  - **Notify** (Értesítés): A kliens feliratkozik (Subscribe). A szerver csak változás esetén küld csomagot, ami energiatakarékos és gyors
- **A Kliens felelőssége:** Ő vezérli a folyamatot: ő keres, ő kapcsolódik, és ő dönt arról, hogy melyik adatra tart igényt

# BLE GATT Kliens Adatkapcsolat Felépítése (ESP32-C3 / NimBLE)

```
// 1. Csatlakozás a talált eszközhöz
NimBLEClient* pClient = NimBLEDevice::createClient();
if (pClient->connect(advDevice)) {
    Serial.println("OK: Csatlakozva.");

    // 2. Szolgáltatás (Service) keresése UUID alapján
    NimBLERemoteService* pSvc = pClient->getService(NimBLEUUID("181A"));

    if (pSvc) {
        Serial.println("OK: Környezeti Szolgáltatás megvan.");

        // 3. Konkrét Jellemző (Characteristic) keresése
        // feltételezzük, hogy van hőmérséklet (0x2A6E)
        NimBLERemoteCharacteristic* cT = pSvc->getCharacteristic(NimBLEUUID("2A6E"));

        // 4. Feliratkozás (Notify) az adatra
        if (cT && cT->canNotify()) {
            cT->subscribe(true, notifyCB);
            Serial.println("OK: Feliratkozás kész.");
        }
    }
}
```



# A GATT_client.ino program vázlata

- **Azonosítók** (UUID-k):
  - **181A**: A környezeti adatok szolgáltatásának szabványos azonosítója.
  - **2A6E, 2A6F, 2A6D:** A hőmérséklet, páratartalom és nyomás egyedi „fiókjai”
- **Eseménykezelők** (Callbacks):
  - *ScanCallbacks:* ha megtalálja a C3-BME280 nevű eszközt, leállítja a keresést
  - *ClientCallbacks:* Kezeli a kapcsolat állapotát, szakadás esetén keresés újraindítás
- **Kapcsolódás** (connectToServer): Kiépíti a kapcsolatot, majd sorban „feliratkozik” mindhárom adatra (*subscribe*). Innentől a szerver automatikusan küldi a frissítéseket
- **Adatfeldolgozás** (notifyCB): ha a szerver adatot küld, ez a függvény fogadja és számolja át a bájtokat fizikai mennyiségekké (pl. osztás 100-zal) és frissíti az OLED kijelzőt
- **Életmentő logika** (loop): Folyamatosan ellenőrzi a kapcsolatot. Ha az megszakad és a rendszer nem szkennel, kényszeríti az új keresést, hogy a rendszer soha ne ragadjon „fagyott” állapotban.

# GATT_client.ino
## 5/1. oldal

```
#include <NimBLEDevice.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#include <Wire.h>
#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 32
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, -1);
static const char* SERVICE_UUID = "181A";    // Enviroment sensing service UUID
static const char* TEMP_UUID    = "2A6E";    // Temperature characteristic UUID
static const char* HUM_UUID     = "2A6F";    // Humidity characteristic UUID
static const char* PRES_UUID    = "2A6D";    // Pressure characteristic UUID

static const NimBLEAdvertisedDevice* advDevice;
static bool doConnect = false;               // True if server found
static bool connected = false;               // True if connected to server

float t = 0, h = 0, p = 0;
String statusMsg = "Inditas...";             // Status message

void updateOLED() {                          // Refresh screen
    display.clearDisplay();     display.setCursor(0,0);
    display.setTextSize(1);     display.setTextColor(1);
    display.println(statusMsg);
    display.printf("T:%.2f C\nH:%.1f %%\nP:%.1f hPa", t, h, p);
    display.display();
}
```

# GATT_client.ino
## 5/2. oldal

```
class ClientCallbacks : public NimBLEClientCallbacks {
    void onConnect(NimBLEClient* pClient) override {
        connected = true;
        statusMsg = "Kapcsolodva";
        updateOLED();
    }

    void onDisconnect(NimBLEClient* pClient, int reason) override {
        connected = false;
        statusMsg = "Szakadt - Ujra scan";
        updateOLED();
        // Szakadás után azonnal indítjuk a végtelen szkennelést
        NimBLEDevice::getScan()->start(0, false, true);
    }
} clientCallbacks;

class ScanCallbacks : public NimBLEScanCallbacks {
    void onResult(const NimBLEAdvertisedDevice* advertisedDevice) override {
        if (advertisedDevice->isAdvertisingService(NimBLEUUID(SERVICE_UUID)) ||
            advertisedDevice->getName() == "C3-BME280") {
            NimBLEDevice::getScan()->stop();
            advDevice = advertisedDevice;
            doConnect = true;
        }
    }
} scanCallbacks;
```

# GATT_client.ino 5/3. oldal

```
bool connectToServer() {
    NimBLEClient* pClient = nullptr;
    if (NimBLEDevice::getCreatedClientCount()) {
        pClient = NimBLEDevice::getClientByPeerAddress(advDevice->getAddress());
        if (pClient) { if (!pClient->connect(advDevice, false)) return false;
        } else { pClient = NimBLEDevice::getDisconnectedClient(); }
    }
    if (!pClient) {
        pClient = NimBLEDevice::createClient();
        pClient->setClientCallbacks(&clientCallbacks, false);
        pClient->setConnectionParams(12, 12, 0, 150);
        pClient->setConnectTimeout(5 * 1000);
        if (!pClient->connect(advDevice)) { NimBLEDevice::deleteClient(pClient); return false; }
    }
    NimBLERemoteService* pSvc = pClient->getService(SERVICE_UUID);
    if (pSvc) {
        NimBLERemoteCharacteristic* cT = pSvc->getCharacteristic(TEMP_UUID);
        NimBLERemoteCharacteristic* cH = pSvc->getCharacteristic(HUM_UUID);
        NimBLERemoteCharacteristic* cP = pSvc->getCharacteristic(PRES_UUID);
        if (cT && cT->canNotify()) cT->subscribe(true, notifyCB);
        if (cH && cH->canNotify()) cH->subscribe(true, notifyCB);
        if (cP && cP->canNotify()) cP->subscribe(true, notifyCB);
        return true;
    }
    return false;
}
```

**Service discovery:**
a kliens megkeresi a jellemzőket

**Subscribe:**
Feliratkozás a Notify üzenetekre

```
//------------------------------------------------------------
// Notify callback függvény
// A szerverről érkező adatokat itt fogadjuk és használjuk fel
//------------------------------------------------------------
void notifyCB(NimBLERemoteCharacteristic* pRemoteCharacteristic, uint8_t* pData,size_t length,bool isNotify) {
    std::string uuid = pRemoteCharacteristic->getUUID().toString();
    if (uuid.find("2a6e") != std::string::npos) t = (*(int16_t*)pData) / 100.0;
    else if (uuid.find("2a6f") != std::string::npos) h = (*(int16_t*)pData) / 100.0;
    else if (uuid.find("2a6d") != std::string::npos) p = (*(uint32_t*)pData) / 10.0;
    updateOLED();
}

void setup() {
     Wire.begin(8, 9);
    if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) for(;;);
    NimBLEDevice::init("C3-Client");
    NimBLEScan* pScan = NimBLEDevice::getScan();
    pScan->setScanCallbacks(&scanCallbacks, false);
    pScan->setInterval(100);
    pScan->setWindow(100);
    pScan->setActiveScan(true);
    pScan->start(0, false);      // START 0: Végtelen szkennelés indítása
    statusMsg = "Szkenneles...";
    updateOLED();
}
```

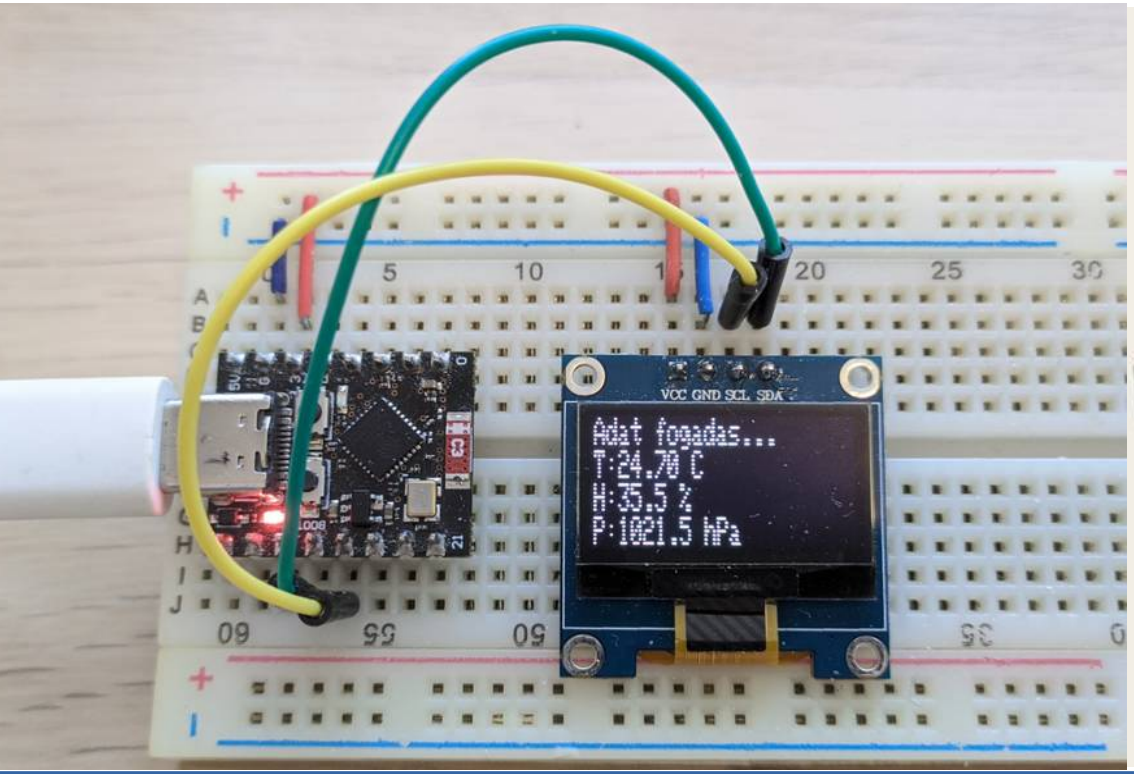
**GATT_client.ino 5/4. oldal**

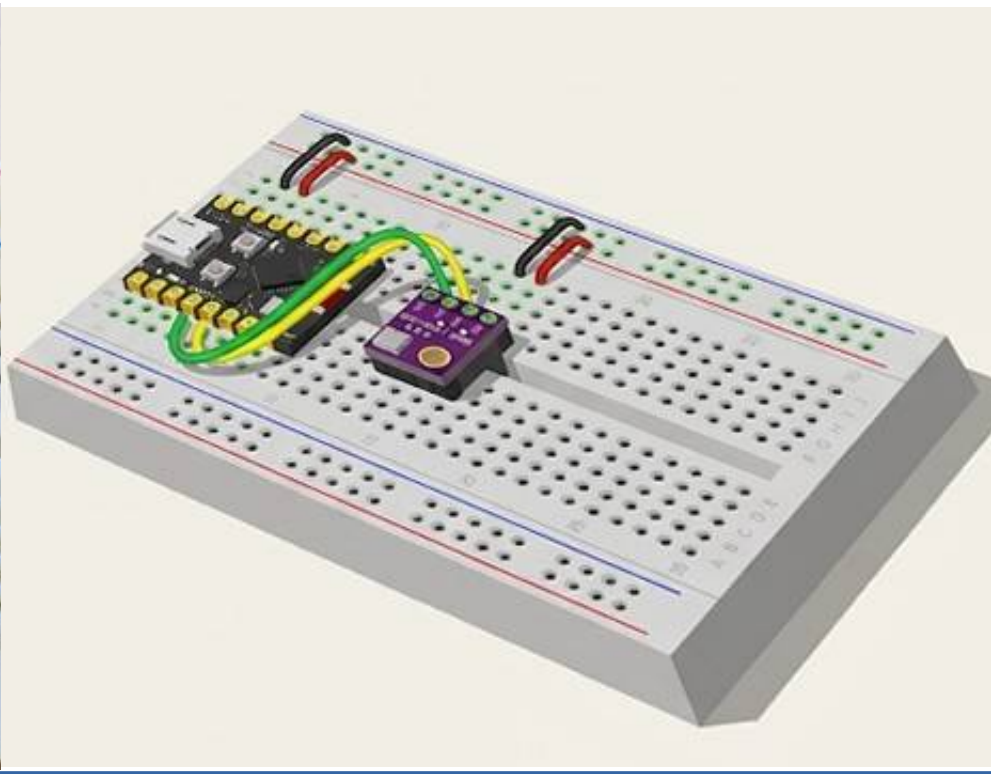
```
void loop() {
    // 1. Csatlakozás vezérlése
    if (doConnect) {
        doConnect = false;
        if (connectToServer()) {
            statusMsg = "Adat fogadas...";
        } else {
            statusMsg = "Hiba - Scan ujra";
            NimBLEDevice::getScan()->start(0, false, true);
        }
        updateOLED();
    }

    // 2. ÉLETMENTŐ LOGIKA: Ha nincs kapcsolat és nem szkennel, indítsa újra!
    // Ez kezeli azt, ha a rádió elakadna vagy időtúllépés történne
    if (!connected && !doConnect) {
        NimBLEScan* pScan = NimBLEDevice::getScan();
        if (!pScan->isScanning()) {
            pScan->start(0, false, true);
        }
    }
    delay(200); // Ne terheljük túl a CPU-t, de reagáljon gyorsan
}
```

**GATT_client.ino 5/5. oldal**

# GATT_client.ino futási eredménye

- Bár a program a 128 x 32 képpont felbontású SSD1306 kijelzőhöz készült, láthatóan a 12 x 64 képpont felbontású kijelzőn is jól olvashatók a feliratok



# Az ESP32 C3 Super Mini kártya kivezetései


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