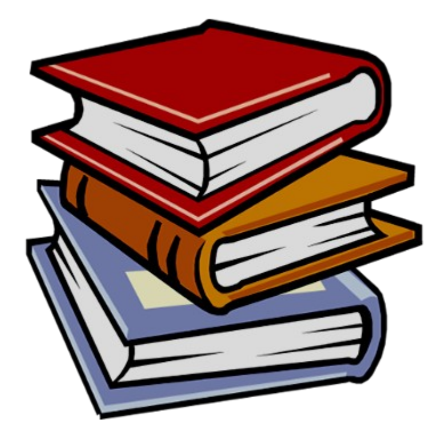
ESP32 mikrovezérlők programozása Arduino környezetben

12. ESP32 Bluetooth kommunikáció

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



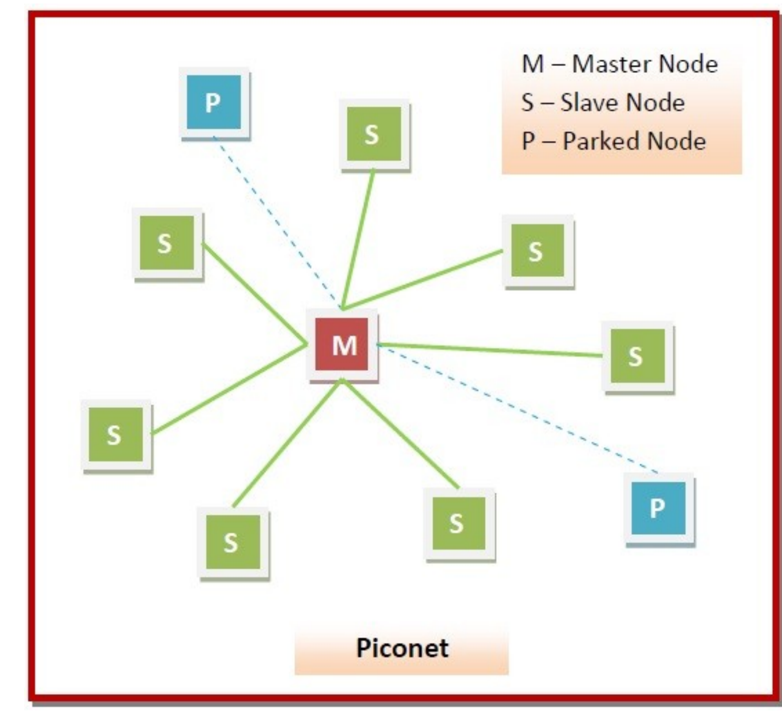
- **Mintaprojektek**
- Timothy Woo: **ESP32 BLE + Android + Arduino IDE = AWESOME**
- **Szoftver segédlet**
  - Kai Morich : Serial Bluetooth Terminal
  - Nordic Semiconductor ASA: nRF Connect for Mobile
  - Fabio Durigon: OLED SSD1306 - SH1106 library
  - Adafruit: BME280 sensor library

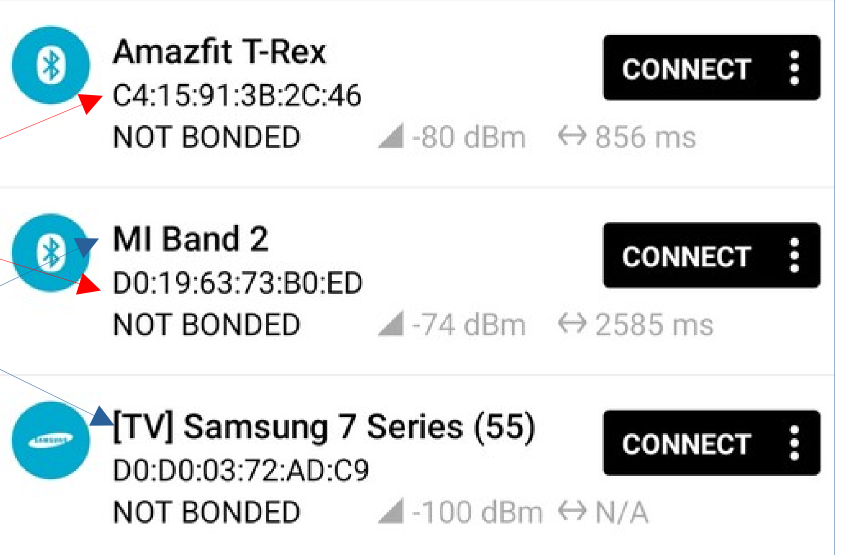
Mi a ?

- 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** natív **Bluetooth** támogatással (4.2 verzió) rendelkezik

Bluetooth piconet

- A **Bluetooth** kommunikáció egy ***mester*** és egy ***szolga*** csomópont között egy az egyhez vagy egy a többhöz módon történhet. A szolgák között azonban nem történik közvetlen kommunikáció
- A ***mester*** csomópont az elsődleges állomás, amely a hozzá csatlakozó ***szolga*** csomópontok (legfeljebb 7 db lehet aktív) kis hálózatát kezeli
- Minden állomás, legyen az ***master*** vagy ***slave,*** egy 48 bites rögzített eszközcímhez van társítva (BD_ADDR)
- Az eszközcímekhez egy szimbolikus név (*display name*) is társítható, ami nem egyedi, s csak BD_ADDR azonosít





Bluetooth profilok és protokollok

- Az alsóbb szinteken a **Bluetooth** a partnerek közötti adatcseréről gondoskodik. A Bluetooth azonban sokkal többet jelent, mint egyszerű adatcsere
- A több gyártó által épített *eszközök közötti együttműködés* érdekében magasabb szintű protokollokat, úgynevezett **profilokat** határoztak meg. Ezek határozzák meg, hogy „mit" továbbítson a **Bluetooth** egy adott eszközfunkció megvalósításához.
- Néhány profil amellyel gyakran találkozhatunk:
  - **HSP** – Head Set Profile (pl. Bluetooth fülhallgató/mikrofon)
  - **HFP** – Hands Free Profile (pl. autós kihangosító)
  - **HID** – Human Interface Device (pl. billentyűzet vagy egér)
  - **SPP** – Serial Port Profile (pl. virtuális soros port)
  - **A2DP** – Advanced Audio Distribution profile (pl. Bluetooth hangszóró).
  - **AVRCP** – Audio Visual Remote Control Profile (pl. távirányító)

BluetoothSerial

- A **BluetoohSerial** programkönyvtár segítségével „klasszikus” **BT** módú soros kommunikációt végezhetünk, a **Serial** osztályhoz hasonló módon
- Az **ESP32 Arduino Core** számos mintapéldát tartalmaz
- A **BluetoothSerial** SerialBT – példányosítás után leggyakrabban az alábbi tagfüggvényeket használjuk:
  - **begin**(*display name*) – inicializálás, a megadott néven lesz látható
  - **available**() – beérkezett üzenet vizsgálata
  - **read**() – a beérkezett üzenet olvasása
  - **write**(*char*) – üzenet küldése
  - **connect**(*name* vagy *address*) – kapcsolódás (master módban)
  - **disconnect**() – kapcsolat bontása

SerialToSerialBT.ino

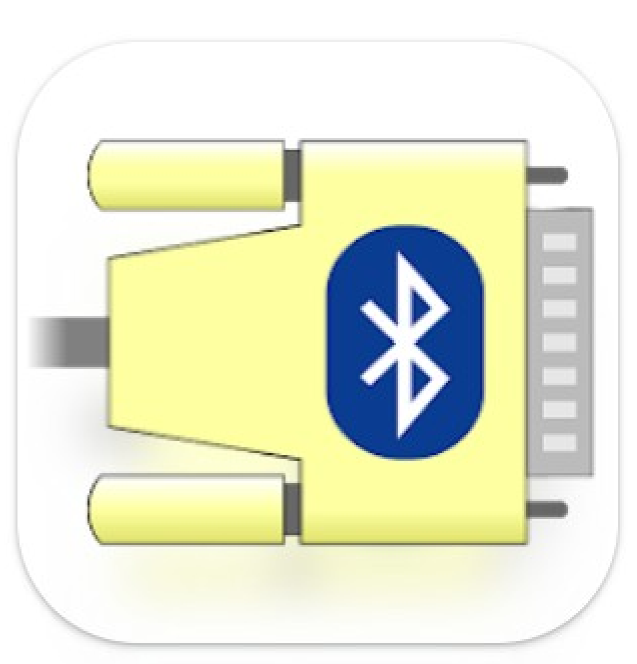
- Összekapcsoljuk a soros és a klasszikus Bluetooth (SPP) portot

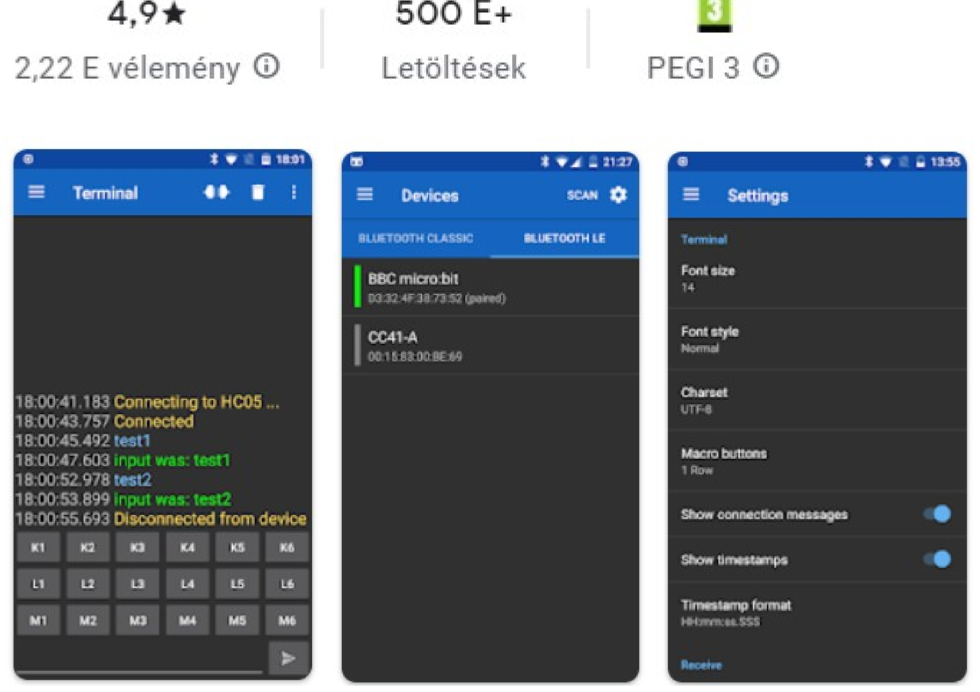
```
#include "BluetoothSerial.h"
#if !defined(CONFIG_BT_ENABLED) || !defined(CONFIG_BLUEDROID_ENABLED)
#error Bluetooth is not enabled! Run `make menuconfig` and enable it
#endif

BluetoothSerial SerialBT;

void setup() {
  Serial.begin(115200);
  SerialBT.begin("ESP32test"); //Bluetooth device name
  Serial.println("The device started, now you can pair it with bluetooth!");
}

void loop() {
  if (Serial.available()) {
    SerialBT.write(Serial.read());
  }
  if (SerialBT.available()) {
    Serial.write(SerialBT.read());
  }
  delay(20);
}
```

ESP32 Arduino Core BluetoothSerial egyik gyári mintapéldája

Serial Bluetooth Terminal

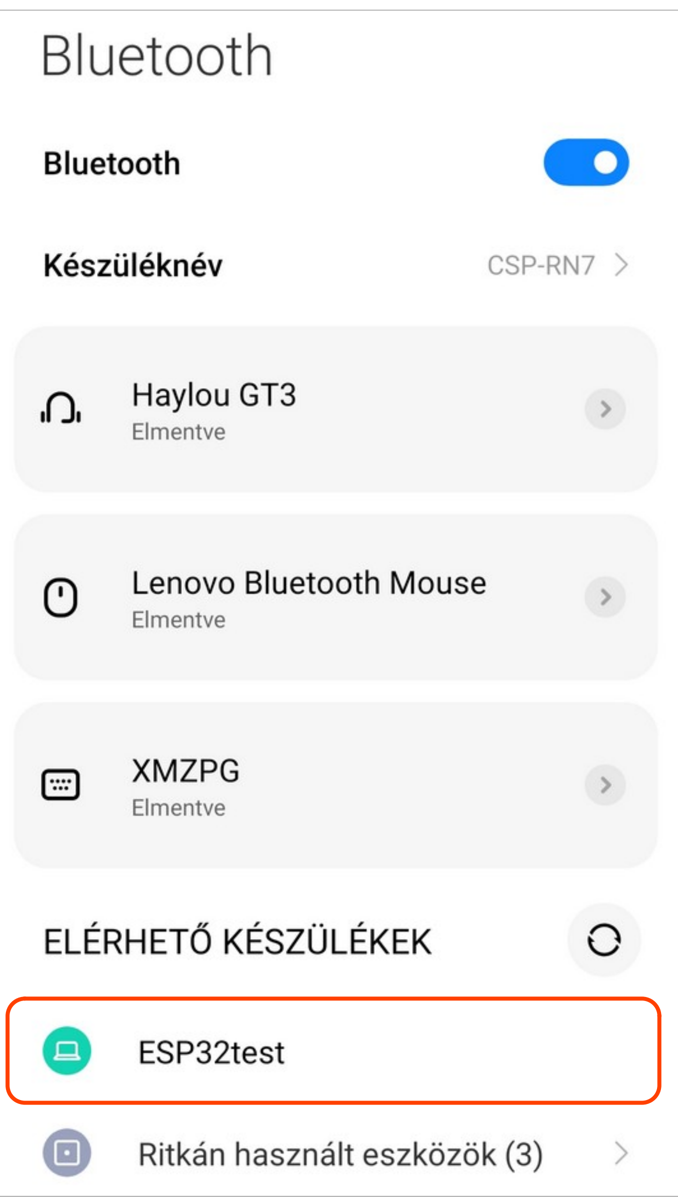
- Kai Morich Serial Bluetooth Terminal Android alkalmazása sor-orientált terminált biztosít, RFCOMM kapcsolatot implementálva az `SPP_UUID = 00001101-0000-1000-8000-00805F9B34FB` profilú klasszikus **Bluetooth SPP** eszközökhöz

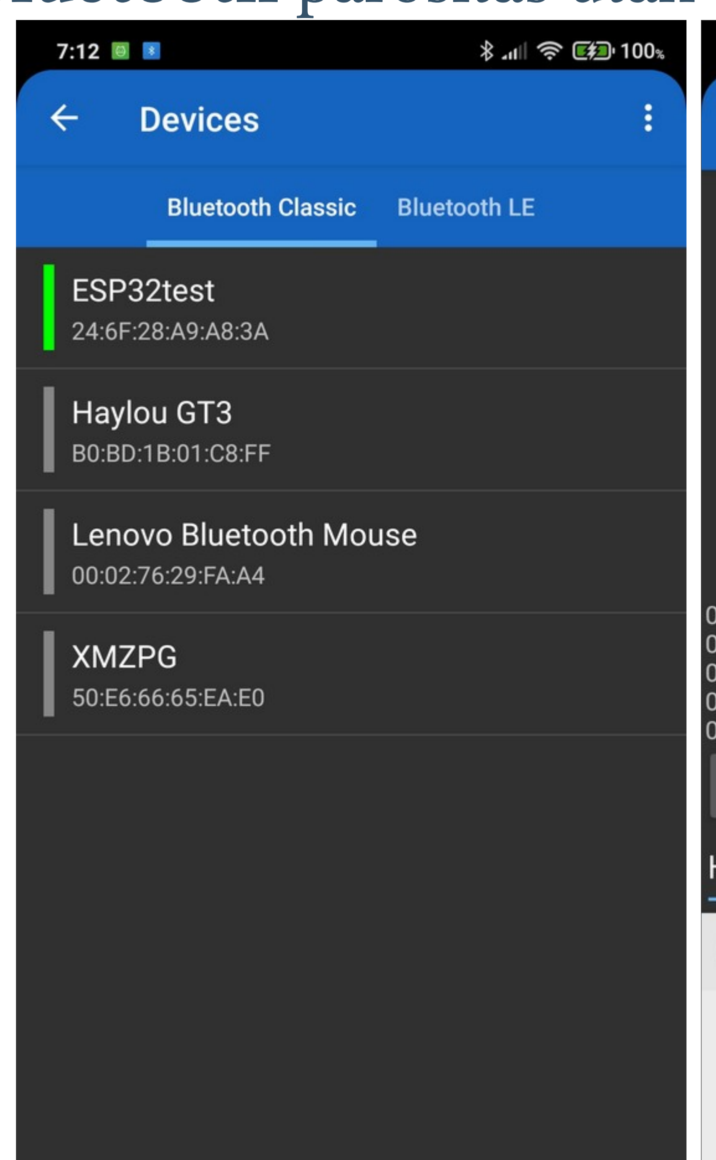


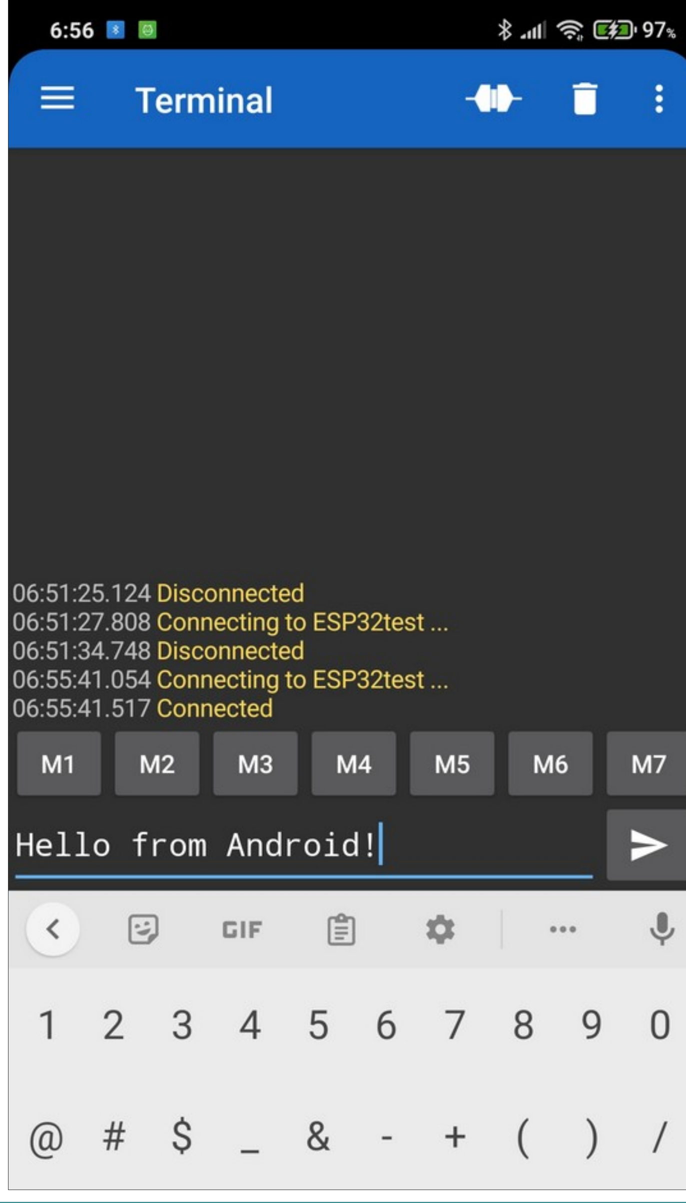


Csatlakozás a Serial Bluetooth Terminálhoz

- Első alkalommal a **Bluetooth** párosítás után csatlakozhatunk

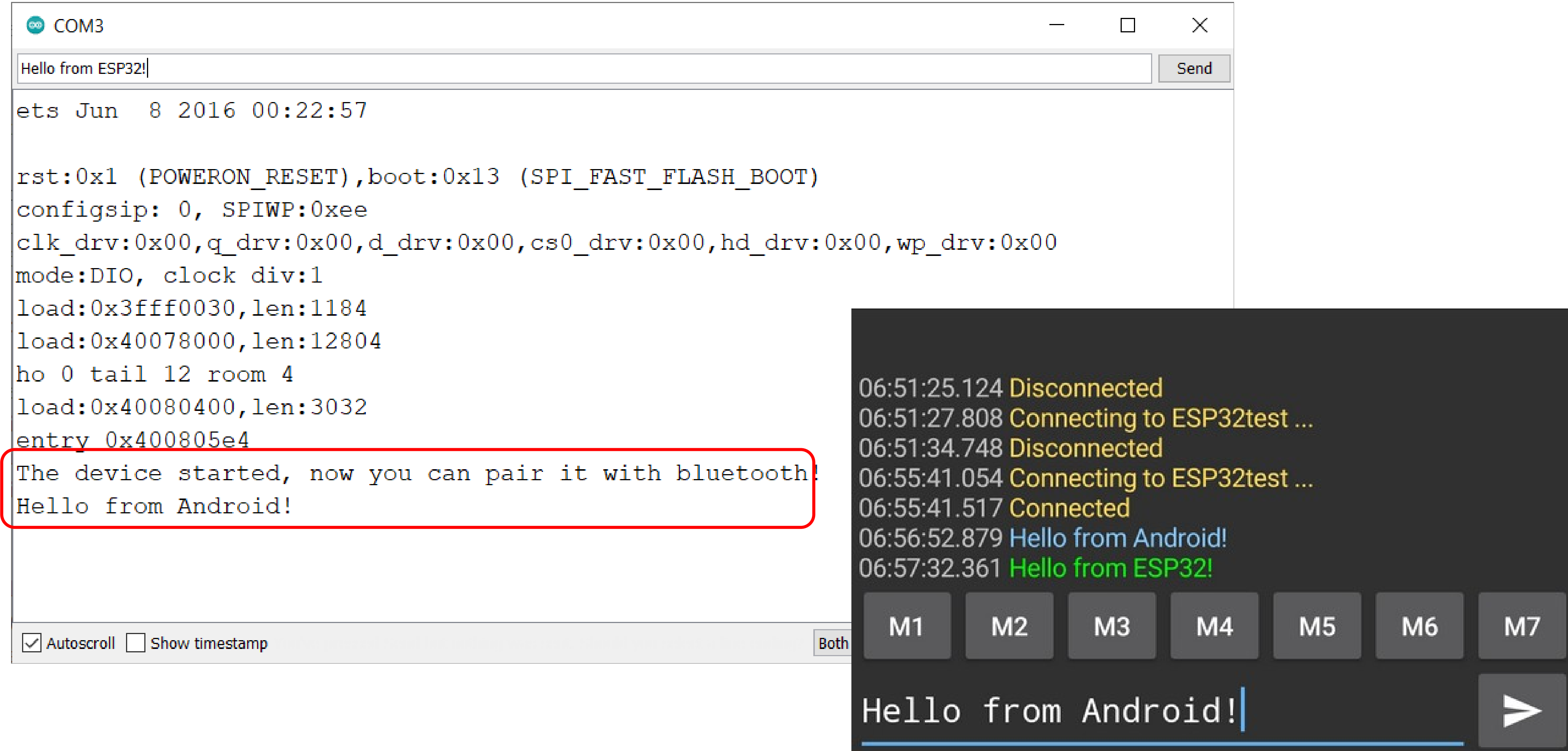






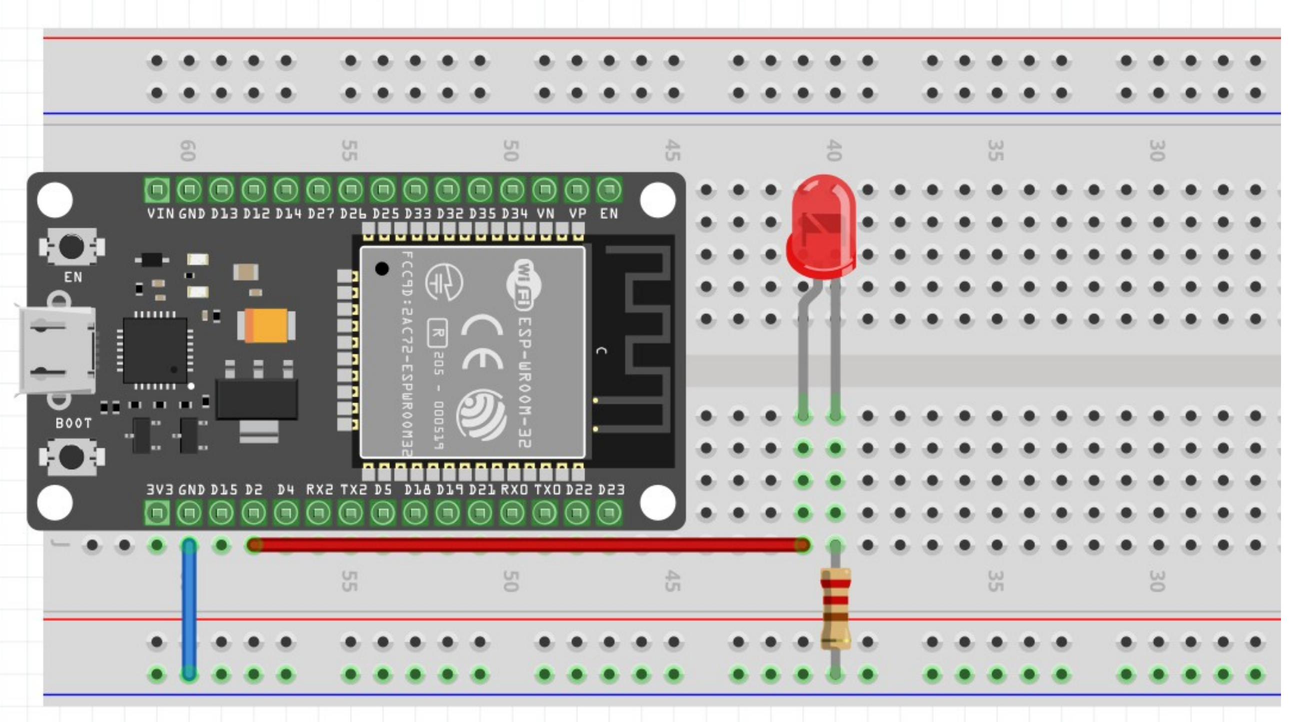
SerialToSerialBT.ino futási eredmény

- A telefonról küldött üzenet megjelenik a terminál ablakában, a soros terminálból küldött üzenetek pedig megjelennek a telefon képernyőjén, tehát működik a kétirányú kapcsolat



BT_serial_ledswitch.ino

- Ebben a programban a **SerialBT** „vezeték nélküli soros porton” beérkező karakterekkel a beépített LED-et vezéreljük (**GPIO2**):
  - Ha a beérkező karakter „1”, akkor a kimenetet magas szintre állítjuk
  - Ha a beérkező karakter „0”, akkor a kimenetet alacsony szintre állítjuk
- Természetesen a LED helyett egy komolyabb eszközt is vezérelhetünk egy relé vagy kapcsolótranzisztor közbeiktatásával



BT_serial_ledswitch.ino

```
#include "BluetoothSerial.h"

BluetoothSerial SerialBT;
int relayPin = 2;

void processReceivedValue(char command) {
  if (command == '1') {
    digitalWrite(relayPin, HIGH);
  } else if (command == '0') { digitalWrite(relayPin, LOW); }
}

void setup() {
  Serial.begin(115200);
  pinMode(relayPin, OUTPUT);
  digitalWrite(relayPin, LOW);
  if (!SerialBT.begin("ESP32")) {
    Serial.println("An error occurred initializing Bluetooth");
  } else { Serial.println("Bluetooth initialized"); }
}

void loop() {
  while (SerialBT.available()) {
    char command = SerialBT.read();
    Serial.println(command);
    processReceivedValue(command);
  }
  delay(50);
}
```

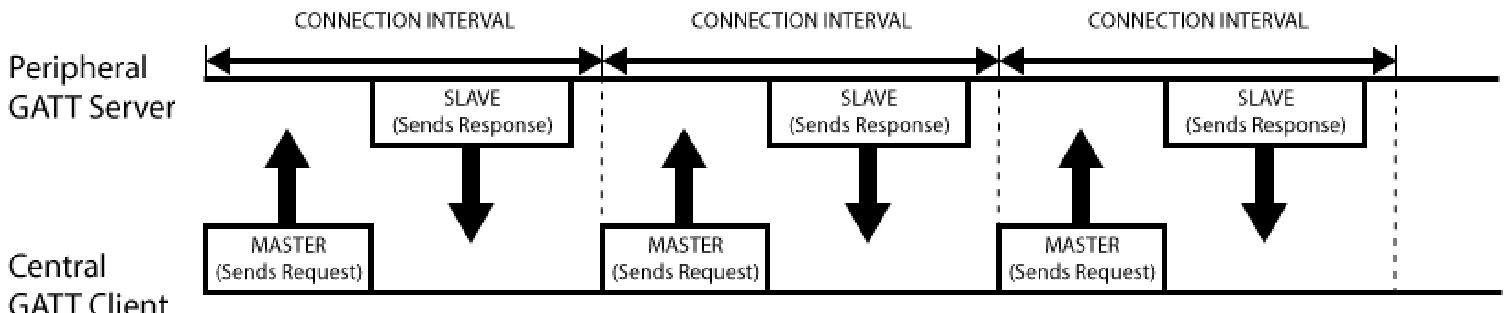
BLE = Bluetooth Low energy

- **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)

Adv Data
Peripheral
Central

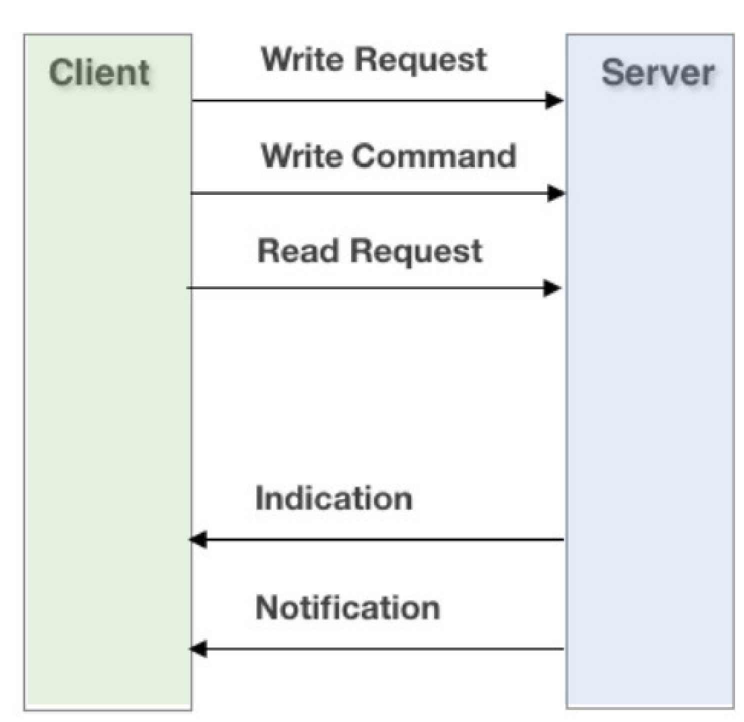
BLE GATT

- Az **Általános Attribútum Protokoll** (**GATT** – Generic Attribute Protocol) mechanizmust biztosít az adatok szabványos átadására, miután az eszközök között már létrejött a kapcsolat
- Gondoljunk a **GATT**-ra úgy, mint az adatküldés és fogadás módja: a kliens eszköz explicit módon **lekérheti** a szerver adatait (az attribútumokat), vagy **push üzeneteket fogad,** amikor pl. a szerveren valamilyen esemény bekövetkezik
- A *perifériát* **GATT-szerver** nevezik, amely a lekérhető attribútumokat (az adatokat), valamint a szolgáltatás- és jellemződefiníciókat tartalmazza
- A **GATT-kliens** (a telefon/számítógép stb.) az az eszköz, amely kéréseket küld ennek a szervernek. Minden tranzakciót a *kliens* indít, amely választ kap a másodlagos eszköztől, a *kiszolgálótól*

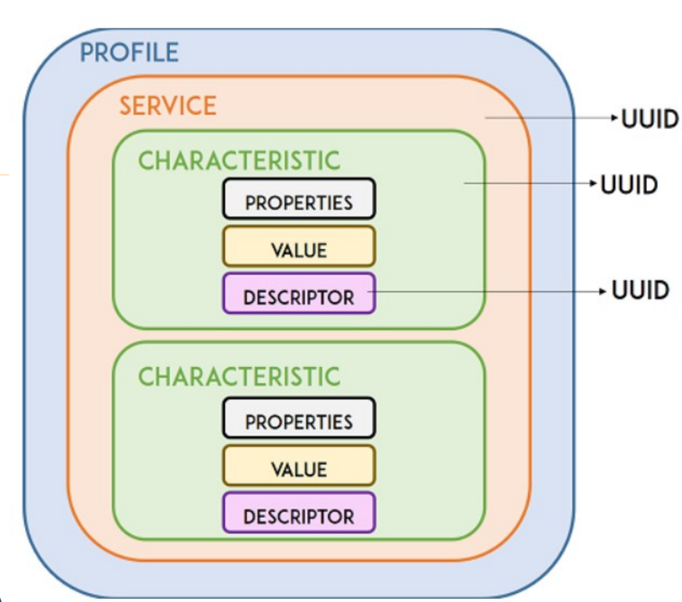


Kliens – szerver kapcsolat

- A szerver és a kliens közötti kapcsolat az alábbiakban látható:
- **A kliens úgy küld adatot** a szervernek, hogy adatokat ír a szerverre, erre a *write request* vagy a *write command* egyaránt használható, de csak az *írás kérés* használata esetén kapunk visszajelzést
- **A szerver adatokat küld a kliensnek** úgy, hogy *jelzést* (indication) vagy *értesítést* (notification) küld a szervernek a kliens
- Az egyetlen különbség a jelzés és az értesítés között az, hogy megerősítés csak indikáció használatakor kérhető
- **A kliens lekérhet adatokat a szerverről** *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 tulajdonság** szabja meg, hogy a *jellemzővel* milyen művelet végezhető

UUID (Universally Unique Identifier)

- Minden *szolgáltatásnak, jellemzőnek* és *leírónak* van egy egyedi 128 bites (16 bájt) UUID azonosítója (Universally Unique Identifier)
- Általában hexadecimálisan írják ezeket (1 bájt 2 hexadecimális számjegynek felel meg), 4-2-2-2-6 (byte) formátumban. Például:

  `4fafc201-1fb5-459e-8fcc-c5c9c331914b`

- Mivel a 16 bájt sok adat egy szolgáltatás azonosságának leírásához, megadhatjuk az UUID-t 16 bites vagy 32 bites rövidített formában is, de ezek a Bluetooth specifikáció tulajdonosai által tervezett szabványos szolgáltatások számára vannak fenntartva
- Amennyiben az alkalmazásunknak saját UUID-re van szüksége, előállíthatjuk ezt az UUID-generátor webhely használatával

ESP32 BLE Server

- Az ESP32 BLE szerver létrehozásának és elindításának lépései
  - Inicializálunk egy ESP32 BLE eszközt
  - Az eszközön létrehozunk egy szervert
  - A szerveren létrehozunk egy BLE szolgáltatást
  - A szolgáltatáshoz létrehozunk egy (vagy több) jellemzőt
  - A jellemzőhöz opcionálisan BLE leírót definiálhatunk
  - Elindítjuk a BLE szolgáltatást
  - Elindítjuk a felderítést lehetővé tevő hirdetést
- A fentieken kívül *visszahívási függvényeket* rendelhetünk a szerver kapcsolódási események, illetve a jellemzők írás/olvasás funkcióinak kiszolgálására
- Fentiek illusztrálására nézzük most meg az **ESP32 Arduino Core** csomag **BLE_server** nevű mintapéldáját!

BLE_server.ino – 2/1.

```
// Based on Neil Kolban example SampleServer.cpp
// Ported to Arduino ESP32 by Evandro Copercini
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>

#define SERVICE_UUID        "4fafc201-1fb5-459e-8fcc-c5c9c331914b"
#define CHARACTERISTIC_UUID "beb5483e-36e1-4688-b7f5-ea07361b26a8"

void setup() {
  Serial.begin(115200);
  Serial.println("Starting BLE work!");

  BLEDevice::init("Long name works now");
  BLEServer *pServer = BLEDevice::createServer();
  BLEService *pService = pServer->createService(SERVICE_UUID);
  BLECharacteristic *pCharacteristic = pService->createCharacteristic(
                                         CHARACTERISTIC_UUID,
                                         BLECharacteristic::PROPERTY_READ |
                                         BLECharacteristic::PROPERTY_WRITE
                                       );
  pCharacteristic->setValue("Hello World says Neil");
  pService->start();
```

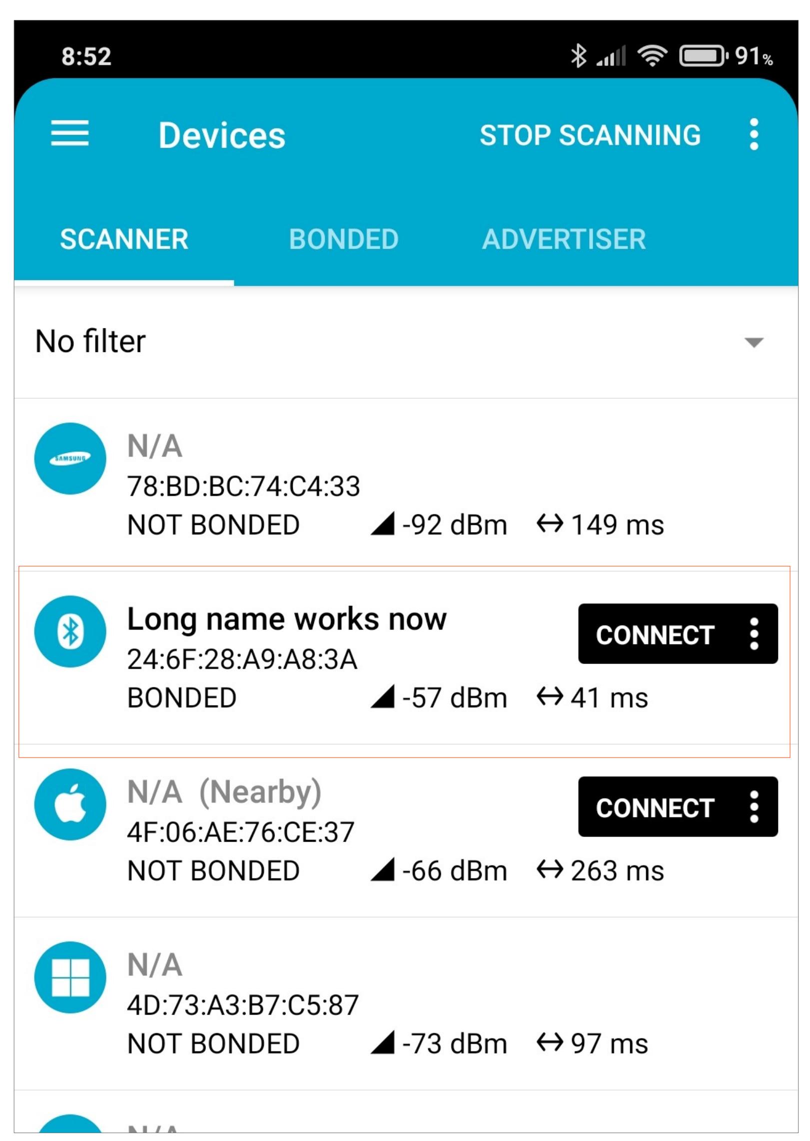
BLE_server.ino – 2/2.

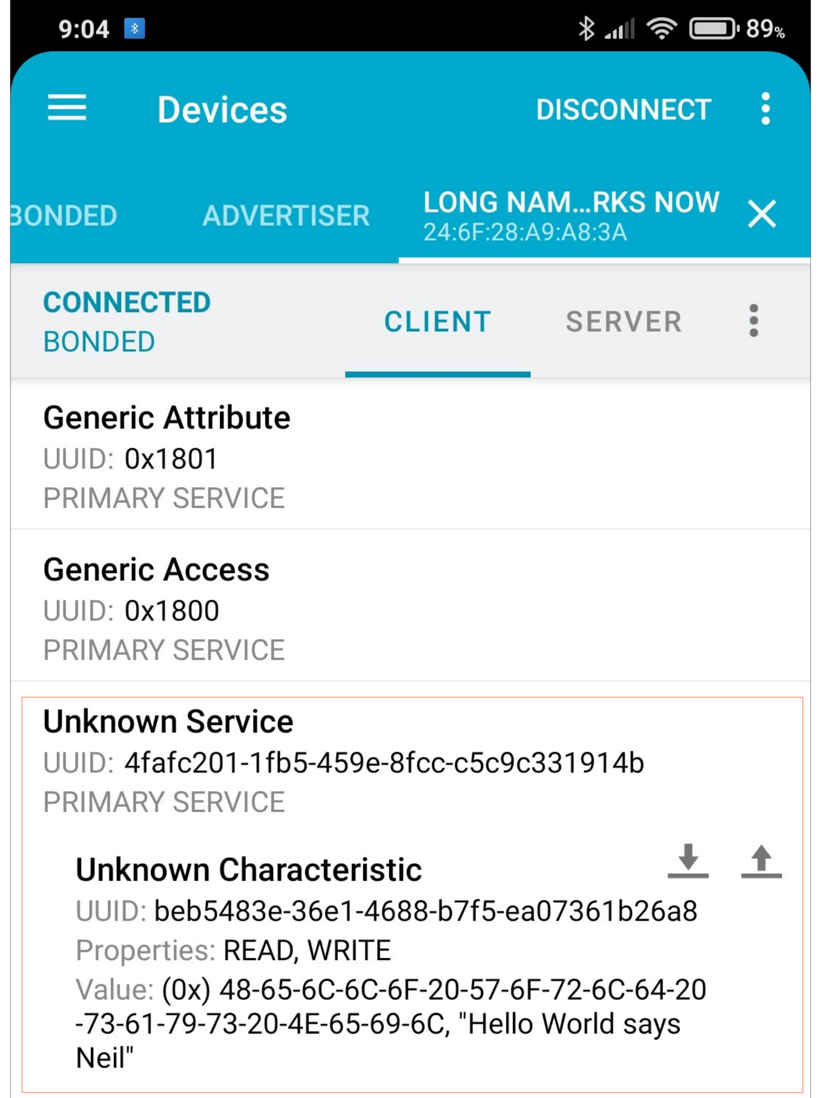
```
  // BLEAdvertising *pAdvertising = pServer->getAdvertising();
  // this still is working for backward compatibility
  BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
  pAdvertising->addServiceUUID(SERVICE_UUID);
  pAdvertising->setScanResponse(true);
  pAdvertising->setMinPreferred(0x06); // help with iPhone connections issue
  pAdvertising->setMaxPreferred(0x12);
  BLEDevice::startAdvertising();
  Serial.println("Characteristic defined! Now you can read it in your phone!");
}

void loop() {
  // put your main code here, to run repeatedly:
  delay(2000);
}
```

- A program működésének vizsgálatához telepítsük a mobiltelefonunkra a Nordic Semiconductor: nRF Connect for Mobile nevű alkalmazását !

BLE_server.ino futási eredménye

- A *Long name…* nevű eszközhöz kapcsolódva az adat írható/olvasható





BLE_scan.ino – 2/1.

- Egy másik ESP32 modulon futtatva, az alábbi program felderíti és kilistázza a közelben lévő, BLE-kompatibilis eszközöket, beleértve a **BLE_server** programot futtató eszközünket is

```
/* Based on Neil Kolban example SampleScan.cpp
   Ported to Arduino ESP32 by Evandro Copercini
*/
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEScan.h>
#include <BLEAdvertisedDevice.h>

int scanTime = 5; //In seconds
BLEScan* pBLEScan;

class MyAdvertisedDeviceCallbacks:
public BLEAdvertisedDeviceCallbacks {
    void onResult(BLEAdvertisedDevice advertisedDevice) {
      Serial.printf("Advertised Device: %s \n",
          advertisedDevice.toString().c_str());
    }
};
```

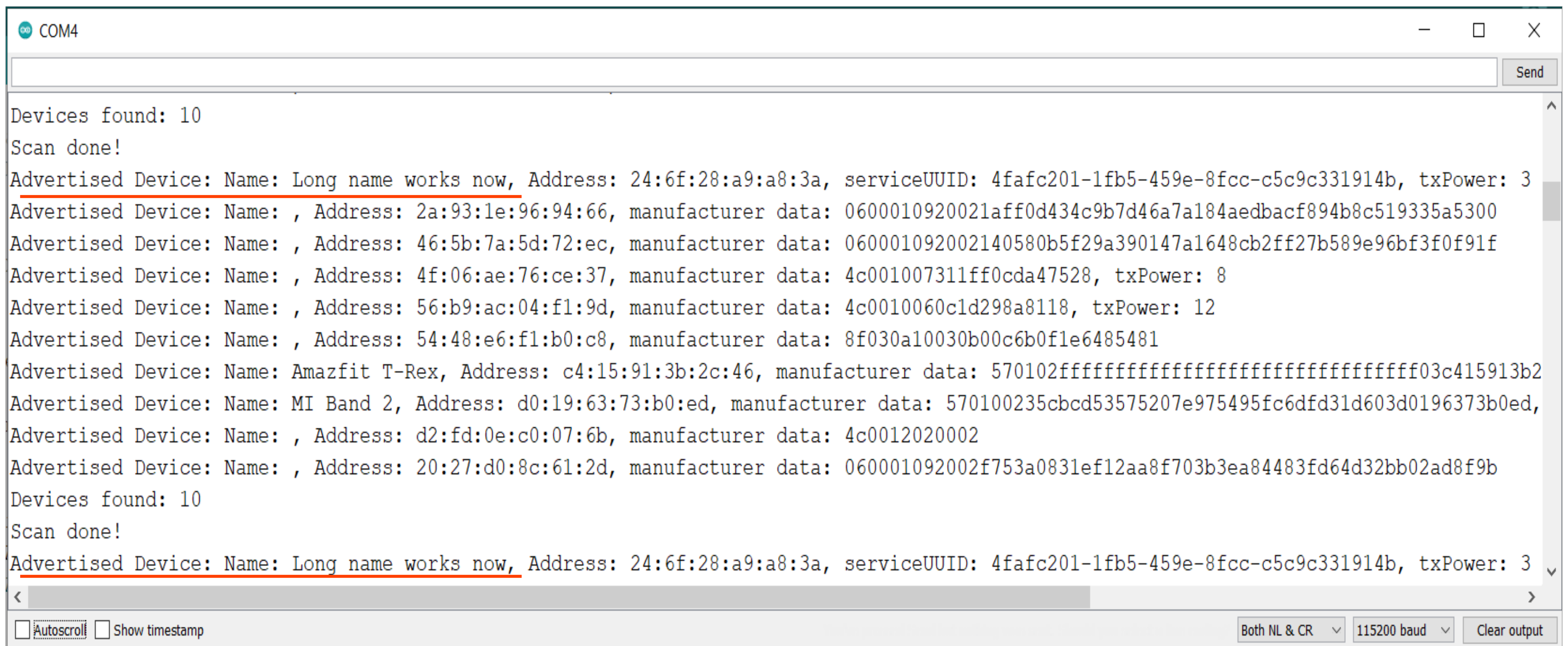
A *scan válasz* eseményt kiszolgáló visszahívási függvényt itt a kiíratással definiálták felül

BLE_scan.ino – 2/1.

```
void setup() {
  Serial.begin(115200);
  Serial.println("Scanning...");
  BLEDevice::init("");
  pBLEScan = BLEDevice::getScan(); //create new scan
  PBLEScan->setAdvertisedDeviceCallbacks(new
                                       MyAdvertisedDeviceCallbacks());
  pBLEScan->setActiveScan(true);   // use more power but get results faster
  pBLEScan->setInterval(100);
  pBLEScan->setWindow(99);         // less or equal setInterval value
}

void loop() {
  BLEScanResults foundDevices = pBLEScan->start(scanTime, false);
  Serial.print("Devices found: ");
  Serial.println(foundDevices.getCount());
  Serial.println("Scan done!");
  pBLEScan->clearResults();        // delete results fromBLEScan buffer
  delay(2000);
}
```

BLE_scan.ino futási eredmény

- A terminálablakban minden pásztázás során megjelenik a felderített eszközök neve (ha van), címe és néhány jellemző adat
- A piros aláhúzással megjelölt sorok a **BLE_server** programot futtató eszközünkre vonatkoznak



BLE_client.ino

- A gyári mintapéldák között található a **BLE_client** példaprogram is, amelynek itt most csak az eredményét mutatjuk be

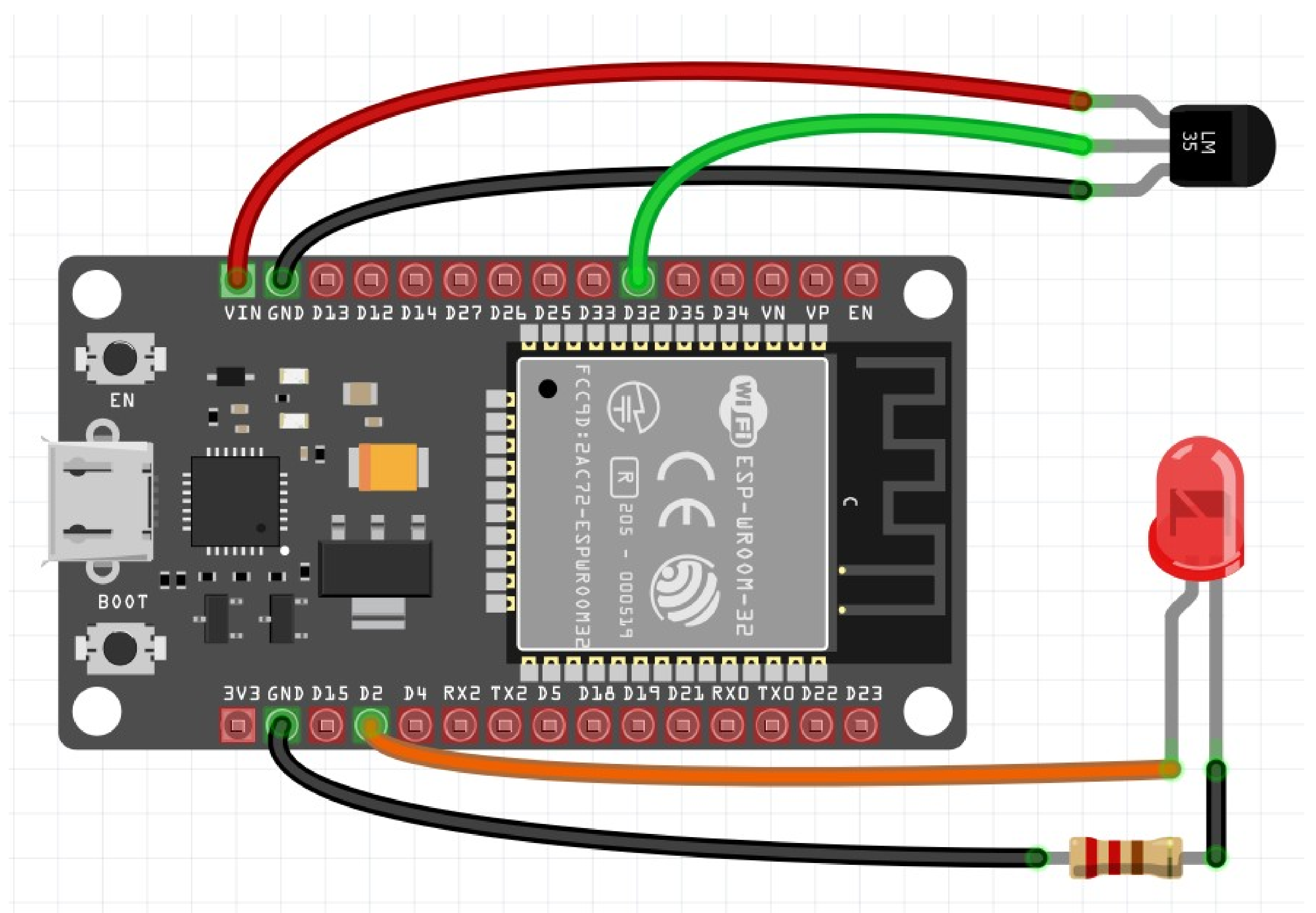
```
Starting Arduino BLE Client application...
BLE Advertised Device found: Name: , Address: 54:48:e6:f1:b0:c8,
                 manufacturer data: 8f030a10030b00c6b0f1e6485481
BLE Advertised Device found: Name: Long name works now, Address: 24:6f:28:a9:a8:3a,
   serviceUUID: 4fafc201-1fb5-459e-8fcc-c5c9c331914b, txPower: 3
Forming a connection to 24:6f:28:a9:a8:3a
 - Created client
 - Connected to server
 - Found our service
 - Found our characteristic
The characteristic value was: Hello World says Neil
We are now connected to the BLE Server.
Setting new characteristic value to "Time since boot: 1"
Setting new characteristic value to "Time since boot: 2"
Setting new characteristic value to "Time since boot: 3"
Setting new characteristic value to "Time since boot: 4"
Setting new characteristic value to "Time since boot: 5"
Setting new characteristic value to "Time since boot: 6"
Setting new characteristic value to "Time since boot: 7"
Setting new characteristic value to "Time since boot: 8"
Setting new characteristic value to "Time since boot: 9"
Setting new characteristic value to "Time since boot: 10"
Setting new characteristic value to "Time since boot: 11"
Setting new characteristic value to "Time since boot: 12"
```

ESP32_BLE_demo

- Egy érdekes **ESP32 BLE** projektet mutatott be Timothy Woo **ESP32 BLE + Android + Arduino IDE = AWESOME** címmel
- A projekt egy **ESP32 BLE** szervert valósít meg, amelyben a *beírt adatokkal* a beépített LED-et (**GPIO2**) vezérelhetjük ('A' hatására bekapcsolja, 'B' hatására kikapcsolja a LED-et)
- A **BLE szerver** bizonyos időközönként értesítést küld a csatlakoztatott kliensnek, hogy új adat áll rendelkezésre
- A projekt egy Android alkalmazást is tartalmaz, amellyel csatlakozhatunk az **ESP32 BLE szerverhez,** megjeleníti a szervertől kapott adatot és egy nyomógombbal vezérelhetjük a LED állapotát
- Az eredeti programot több okból módosítani kellett, továbbá kiegészítettük egy **MCP9700** analóg hőmérővel, így a Celsius fokokban mért aktuális hőmérsékletet küldjük el a kliensnek

Kapcsolási vázlat

- Az analóg hőmérő jele a **GPIO32** analóg bemenetre kötjük
- A **BLE kliens** jelével a **GPIO2** digitális kimenetet vezéreljük



ESP32_BLE_demo.ino – 4/1.

```
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include "LUT.h"        // Lásd: 4. Analóg I/O, analóg szenzorok
#define SERVICE_UUID           "6E400001-B5A3-F393-E0A9-E50E24DCCA9E"
#define CHARACTERISTIC_UUID_RX "6E400002-B5A3-F393-E0A9-E50E24DCCA9E"
#define CHARACTERISTIC_UUID_TX "6E400003-B5A3-F393-E0A9-E50E24DCCA9E"

BLEServer *pServer = NULL;
BLECharacteristic * pTxCharacteristic;
bool deviceConnected = false;
float txValue = 0;
const int ADC_PIN = 32; // Use GPIO number. See ESP32 board pinouts
const int LED = 2;      // Could be different depending on the dev board

class MyServerCallbacks: public BLEServerCallbacks {
    void onConnect(BLEServer* pServer) {
      deviceConnected = true;
    };
    void onDisconnect(BLEServer* pServer) {
      deviceConnected = false;
    }
};
```

ESP32_BLE_demo.ino – 4/2.

```
class MyCallbacks: public BLECharacteristicCallbacks {
    void onWrite(BLECharacteristic *pCharacteristic) {
      std::string rxValue = pCharacteristic->getValue();
      if (rxValue.length() > 0) {
        Serial.println("*********");
        Serial.print("Received Value: ");
        for (int i = 0; i < rxValue.length(); i++) {
          Serial.print(rxValue[i]);
        }
        Serial.println();
        if (rxValue.find("A") != -1) {
          Serial.print("Turning ON!");
          digitalWrite(LED, HIGH);
        }
        else if (rxValue.find("B") != -1) {
          Serial.print("Turning OFF!");
          digitalWrite(LED, LOW);
        }
        Serial.println();
        Serial.println("*********");
      }
    }
};
```

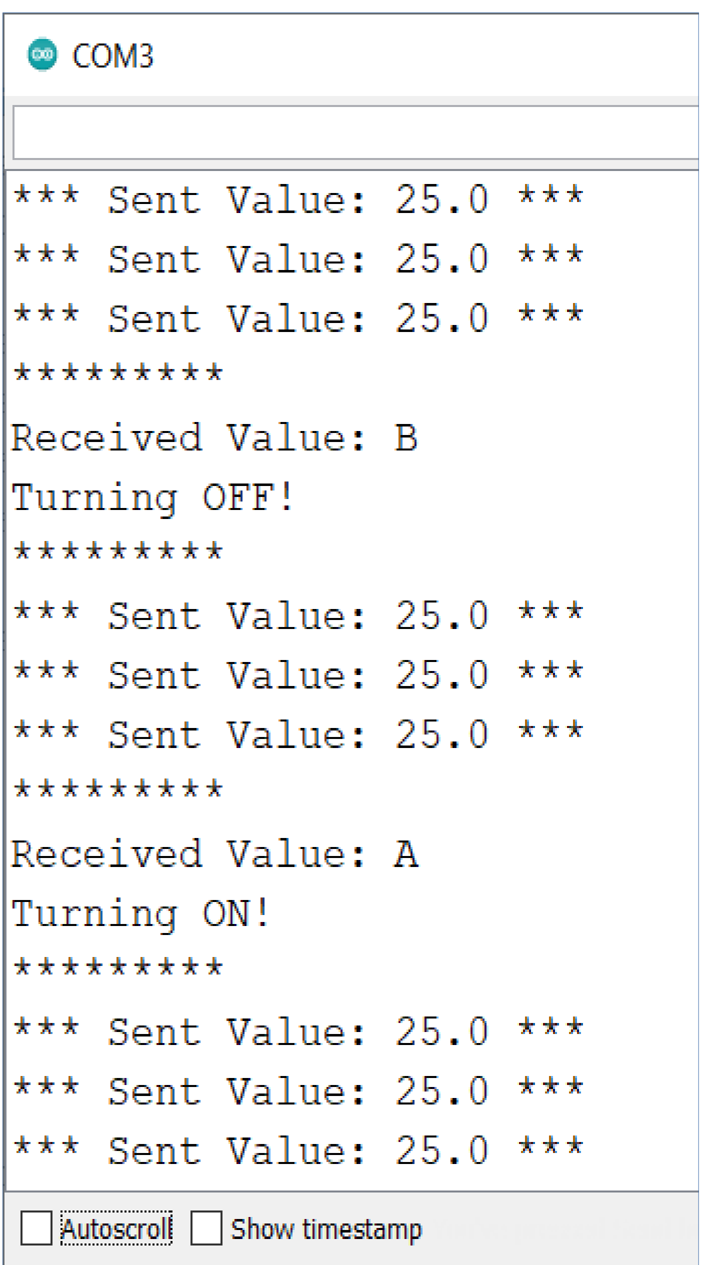
ESP32_BLE_demo.ino – 4/3.

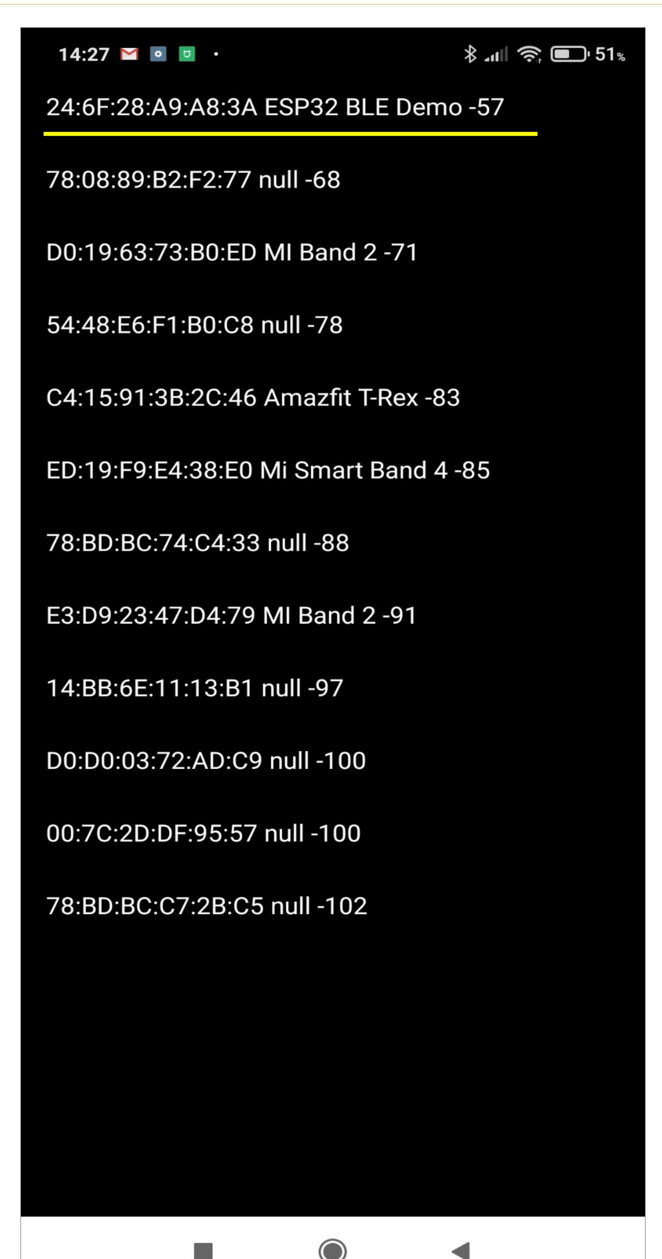
```
void setup() {
  Serial.begin(115200);
  analogReadResolution(12);                   // 12 bites felbontás
  analogSetAttenuation(ADC_11db);             // Méréshatár 0-3,3 V
  pinMode(LED, OUTPUT);

  BLEDevice::init("ESP32 BLE Demo");          // Create BLE device
  pServer = BLEDevice::createServer();        // Create the BLE server
  pServer->setCallbacks(new MyServerCallbacks());

  BLEService *pService = pServer->createService(SERVICE_UUID);

  BLECharacteristic *pRxCharacteristic = pService->createCharacteristic(
      CHARACTERISTIC_UUID_RX, BLECharacteristic::PROPERTY_WRITE );
  pRxCharacteristic->setCallbacks(new MyCallbacks());

  pTxCharacteristic = pService->createCharacteristic(
      CHARACTERISTIC_UUID_TX, BLECharacteristic::PROPERTY_NOTIFY );
  pTxCharacteristic->setValue("Hello!");
  pService->start();                           // Start the service
  pServer->getAdvertising()->start();          // Start advertising
  Serial.println("Waiting a client connection to notify...");
}
```

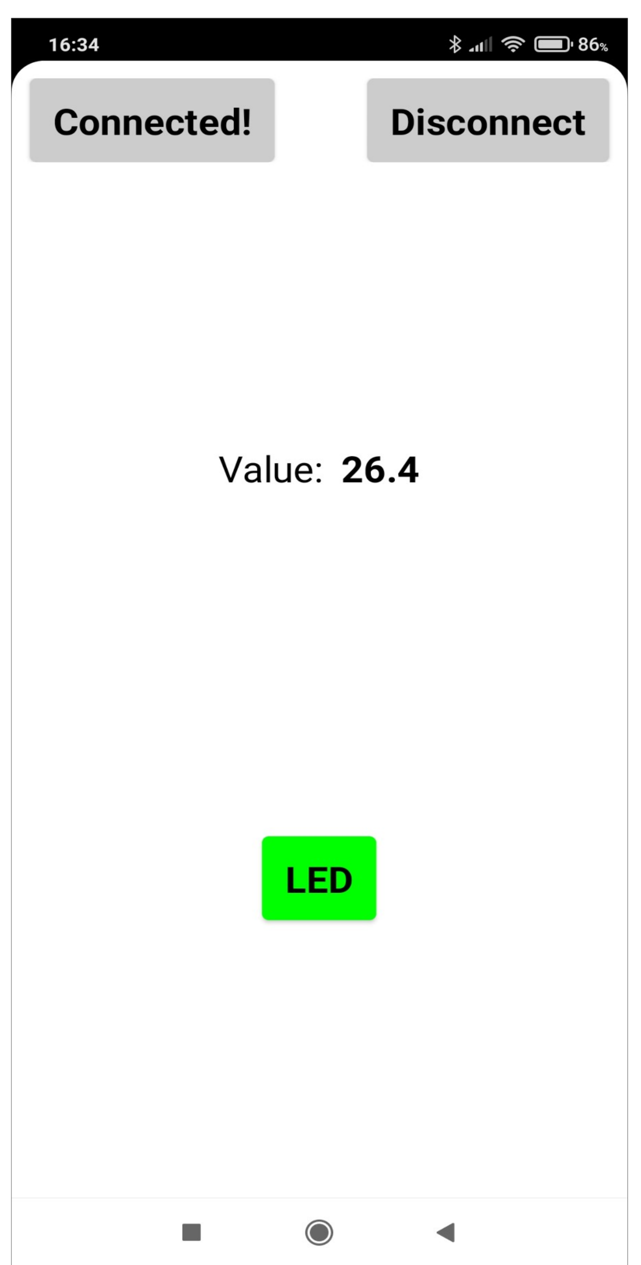
ESP32_BLE_demo.ino – 4/4.

```
void loop() {
  if (deviceConnected) {
    uint32_t rawReading = 0 ;
    for (int i = 0; i < 1024; i++) {              // 1024 mérést átlagolunk
      rawReading += analogRead(ADC_PIN);          // az ADC kiolvasása
    }
    rawReading = rawReading >> 10;                // osztás 1024-gyel

    int calibratedReading = ADC_LUT[rawReading]; // a kalibrált érték
    int milliVolts = calibratedReading*3300/4096;// Átszámítás mV-ra
    float tempC = (milliVolts - 500) / 10.0;      // Átszámítás C fokokra
    char txString[8]; // make sure this is big enough
    dtostrf(tempC, 1, 1, txString);               // float_val, min_width,
                                                  // decimals, char_buffer

    pTxCharacteristic->setValue(txString);
    pTxCharacteristic->notify(); // Send the value to the app!
    Serial.print("*** Sent Value: ");
    Serial.print(txString);
    Serial.println(" ***");
  }
  delay(1000);
}
```

ESP32_BLE_demo futási eredménye







BLE_notify

- Tanulságos lehet megtekinteni az **ESP32 Arduino Core** csomag **BLE_notify** mintapéldáját is
- Ebben a **BLE_server**-hez hasonló *szolgáltatást* és *jellemzőt* definiálunk, de most periodikusan módosítjuk a jellemző értékét és erről értesítjük a felcsatlakozott klienst is – pontosabban csak akkor értesítjük a klienst, ha az megengedi
- Az utóbb említett funkció megvalósításához a jellemzőhöz egy **0x2902** azonosítójú leírót rendelünk
- Az **nRF Connect for Mobile** alkalmazásban látni fogjuk, hogy a leíró is rendelkezik egy 16 bites írható/olvasható adattal, s amíg ennek az értéke nulla, addig nem jönnek az értesítések
- Ha 1-et írunk bele, akkor jönnek az értesítések

BLE_notify.ino – 3/1.

```
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>      // Implementation of the 0x2902 descriptor

BLEServer* pServer = NULL;
BLECharacteristic* pCharacteristic = NULL;
bool deviceConnected = false;
bool oldDeviceConnected = false;
uint32_t value = 0;

#define SERVICE_UUID        "4fafc201-1fb5-459e-8fcc-c5c9c331914b"
#define CHARACTERISTIC_UUID "beb5483e-36e1-4688-b7f5-ea07361b26a8"

class MyServerCallbacks: public BLEServerCallbacks {
    void onConnect(BLEServer* pServer) {
      deviceConnected = true;
    };

    void onDisconnect(BLEServer* pServer) {
      deviceConnected = false;
    }
};
```

BLE_notify.ino – 3/2.

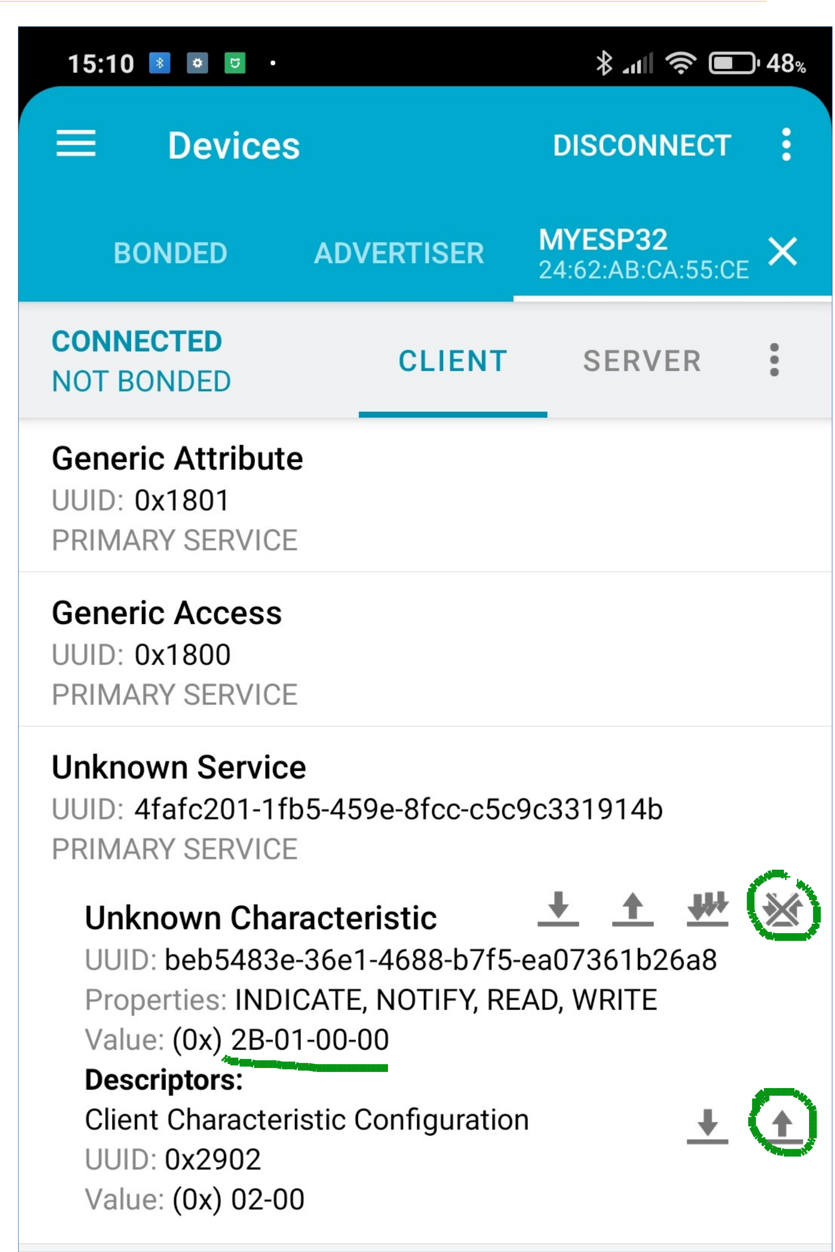
```
void setup() {
  Serial.begin(115200);
  // Create the BLE Device
  BLEDevice::init("myESP32");
  // Create the BLE Server
  pServer = BLEDevice::createServer();
  pServer->setCallbacks(new MyServerCallbacks());

  // Create the BLE Service
  BLEService *pService = pServer->createService(SERVICE_UUID);

  // Create a BLE Characteristic
  pCharacteristic = pService->createCharacteristic(
                      CHARACTERISTIC_UUID,
                      BLECharacteristic::PROPERTY_READ   |
                      BLECharacteristic::PROPERTY_WRITE  |
                      BLECharacteristic::PROPERTY_NOTIFY |
                      BLECharacteristic::PROPERTY_INDICATE
                    );

  pCharacteristic->addDescriptor(new BLE2902()); // Create a BLE Descriptor

  pService->start();    // Start the service
```

BLE_notify.ino – 3/3.

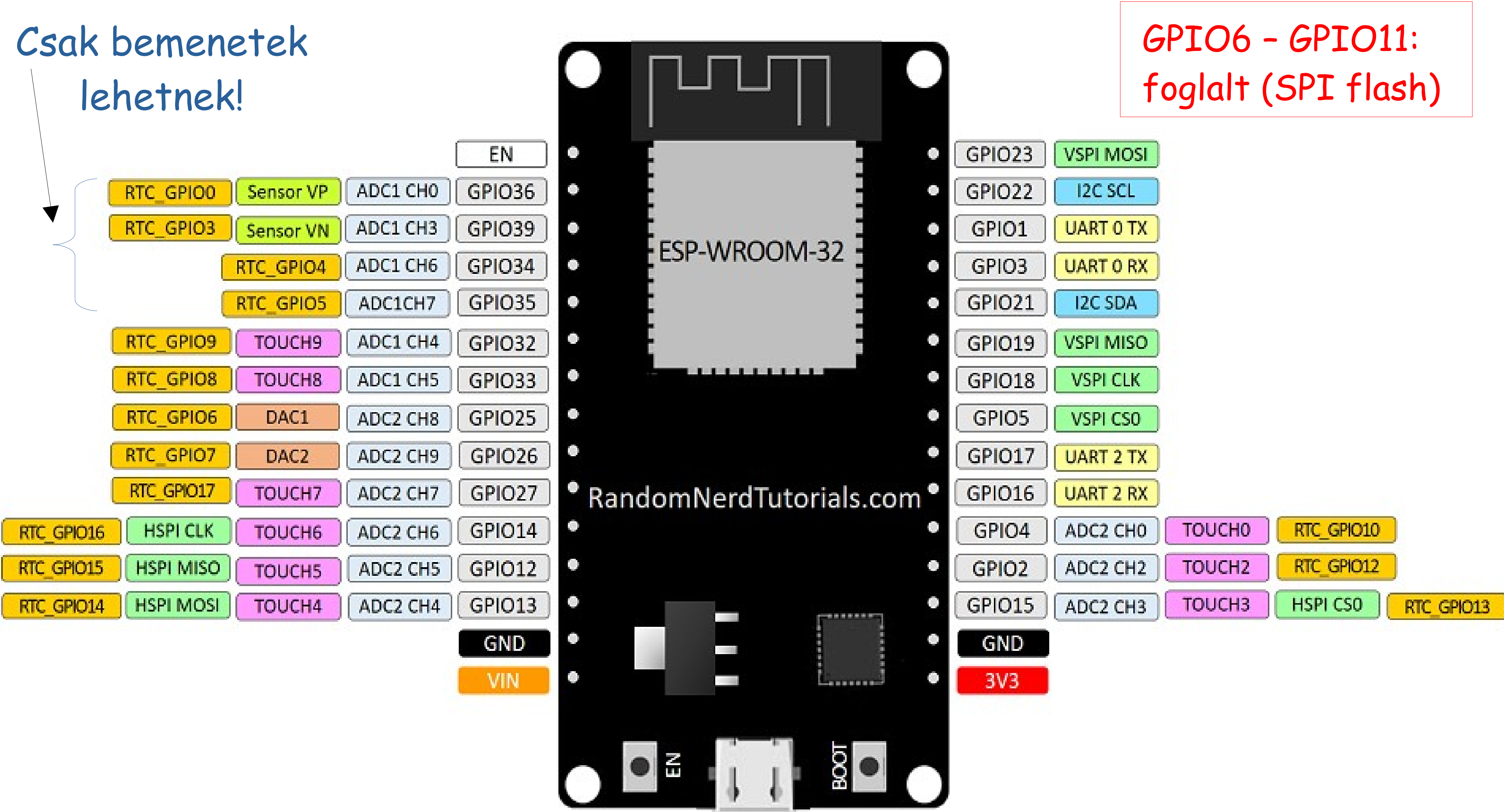
```
  BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
  pAdvertising->addServiceUUID(SERVICE_UUID);
  pAdvertising->setScanResponse(false);
  pAdvertising->setMinPreferred(0x0); // set value to 0x00 to not advertise
  BLEDevice::startAdvertising();
  Serial.println("Waiting a client connection to notify...");
}

void loop() {
    if (deviceConnected) {             // notify changed value
        value = millis()/1000;
        pCharacteristic->setValue((uint8_t*)&value, 4);
        pCharacteristic->notify();
        value++; delay(3);             // don't send packets too frequently
    }
    if (!deviceConnected && oldDeviceConnected) {
        delay(1000); pServer->startAdvertising(); // restart advertising
        Serial.println("start advertising");
        oldDeviceConnected = deviceConnected;
    }
    if (deviceConnected && !oldDeviceConnected) {
        oldDeviceConnected = deviceConnected;
    }
}
```

BLE_notify

- A **0x2902** UUID-jű leíró értéke azt szabályozza, hogy a **Notify** jelzések csak akkor menjenek ki, ha a kliens hajlandó „megnyitni a fülét”
- A leíró alapértelmezett értéke nulla, ami letiltja a küldést
- A kliensnek kell módosítania a leíróhoz tartozó értéket, hogy engedélyezze a küldést
- Az érvényesülő Notify jelzések hatására a kliens kiolvassa a jellemző adatát (itt négy bájt)



A DOIT ESP32 Devkit-1 kártya kivezetései



- **Forrás:** randomnerdtutorials.com/getting-started-with-esp32/

Ellenállás színkódok