

BLE para todos.

Historia de Bluetooth®

El estándar Bluetooth® fue concebido originalmente por el Dr. Jaarp Haartsen en Ericsson en 1994, hace más de 20 años. Lleva el nombre de un renombrado vikingo y rey que unió Dinamarca y Noruega en el siglo X, el rey Harald Gormsson. El Dr. Haartsen fue designado para desarrollar un estándar de conexión inalámbrica de corto alcance que podría reemplazar al estándar RS-232, un estándar de telecomunicaciones por cable que fue concebido en los años 60 y que todavía se usa en la actualidad.

Bluetooth® utiliza lo que se conoce como tecnología de radio de enlace corto. Opera en la banda sin licencia, pero regulada de 2,4 a 2,485 GHz y utiliza radios para comunicarse y establecer conexiones entre dos o más dispositivos. Bluetooth® se basa en el método de espectro ensanchado por salto de frecuencia, este método fue descrito por primera vez en la década de 1940 por la actriz Hedy Lamarr y el compositor George Antheil. Lamarr y Antheil querían crear una forma de evitar que los torpedos guiados por radio se atascaran. Bluetooth® es, en esencia, una red inalámbrica de corto alcance llamada piconet.

En 1994, además de Ericsson, empresas como Intel, Nokia, IBM y Toshiba también tuvieron la idea de un enlace inalámbrico de corto alcance entre dispositivos electrónicos. Lo que estas empresas entendieron en ese momento fue que para crear un enlace inalámbrico de corto alcance que pudiera usarse en diferentes dispositivos electrónicos, había que estandarizar un protocolo para que pudiera aplicarse universalmente. En 1996, esas empresas formaron el Grupo de Interés Especial Bluetooth® (SIG) y finalmente se estableció en 1998. SIG comenzó con solo 5 miembros, al final de su primer año alcanzó los 4.000 miembros y en la actualidad tiene más de 30.000.

El objetivo de Bluetooth® es unir dispositivos al igual que el rey Harald Gormsson unió a las tribus de Dinamarca en un solo reino.

Bluetooth® 1.0 se lanzó alrededor de 1999, la versión 2.0 en 2004, la versión 2.1 en 2007, la versión 3.0 en 2009, la versión 4.0 en 2010, la versión 4.1 en 2013, la versión 4.2 en 2014, la versión 5.0 en 2016 y la versión 5.1 en 2019.

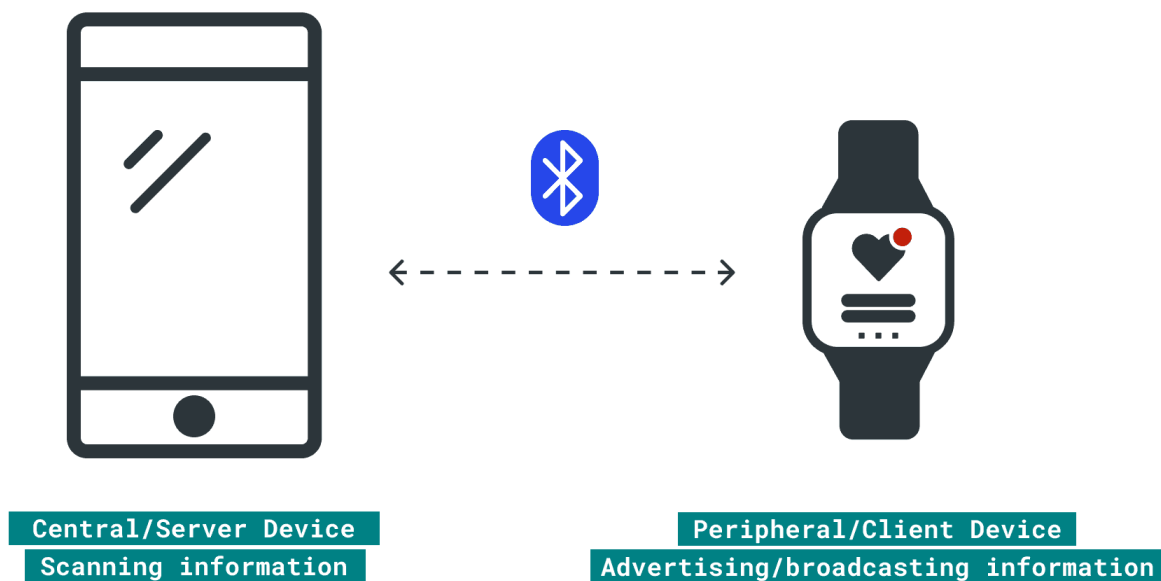
Si busca la especificación Bluetooth® 3.0, encontrará que esta especificación incluye tres modos de trabajo: BR, EDR y HS. Estos tres modos de trabajo son lo que la gente suele llamar, por comodidad, Bluetooth® clásico. En 2010, SIG se fusionó con Wibree, una tecnología inalámbrica desarrollada por Nokia, Nordic Semiconductor y otras empresas cuyo objetivo era encontrar una tecnología de comunicación inalámbrica de bajo consumo para dispositivos electrónicos. SIG cambió el nombre de Wibree a Bluetooth® Low Energy. Bluetooth® Low Energy fue diseñado para reducir significativamente el consumo de energía al reducir la cantidad de tiempo que la radio Bluetooth® está encendida. Tanto el Bluetooth® clásico como el Bluetooth® Low Energy están incluidos desde la especificación Bluetooth® 4.0, pero aquí está la cuestión: Bluetooth® Classic y Bluetooth® Low Energy funcionan de manera diferente y no son compatibles.

Cada modo, Bluetooth® Classic y Bluetooth® Low Energy tienen diferentes métodos de modulación y demodulación de capa física. Esto significa que el Bluetooth® Classic y el Bluetooth® Low Energy no pueden funcionar entre sí. En términos generales, el Bluetooth® clásico se usa principalmente para

aplicaciones de audio (auriculares inalámbricos, por ejemplo), mientras que Bluetooth® Low Energy se ve más a menudo en aplicaciones con restricciones de energía (como dispositivos portátiles y de IoT, por ejemplo).

¿Cómo funciona Bluetooth® de bajo consumo?

Para comprender cómo funciona Bluetooth® Low Energy, debemos hablar sobre las funciones y responsabilidades de dos dispositivos que están conectados a través de Bluetooth®. En cualquier conexión Bluetooth® se juegan dos roles: el central y el periférico. Los dispositivos con una función central también son servidores de llamadas, mientras que los dispositivos con una función periférica también se denominan clientes.



Roles centrales y periféricos en aplicaciones Bluetooth®.

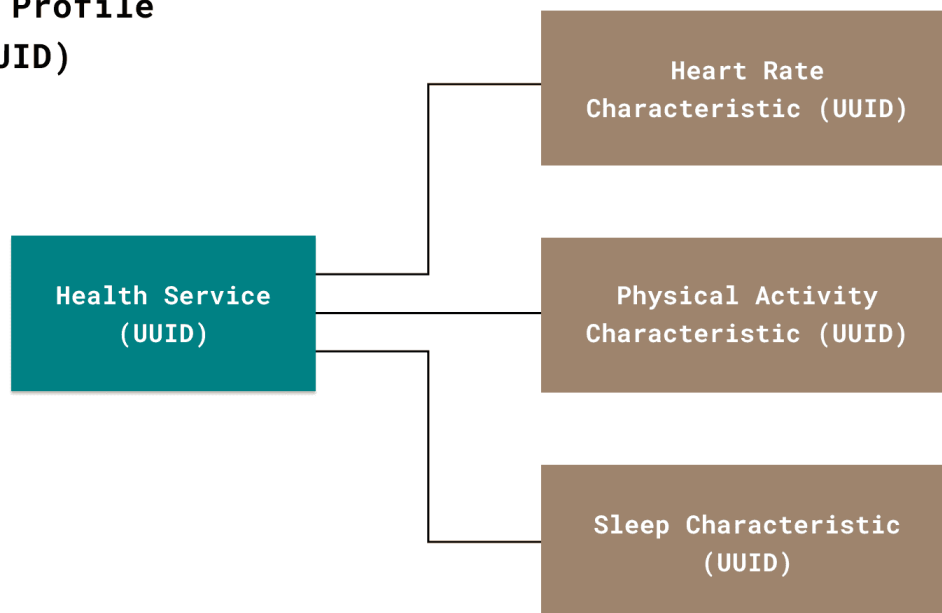
Cuando se establece una conexión Bluetooth®, un dispositivo, el periférico, anunciará o transmitirá información sobre sí mismo a cualquier dispositivo cercano. Al mismo tiempo, otro dispositivo, el central, realizará un escaneo y estará a la escucha de cualquier dispositivo o dispositivos que estén transmitiendo información. Tan pronto como el dispositivo central recoja la información publicitaria del dispositivo periférico, se realizará un intento de conectar el dispositivo periférico. Una vez que se establece una conexión, el dispositivo central interactuará con la información disponible que tiene el dispositivo periférico. Este intercambio de información se realiza mediante lo que se conoce como servicios.

Servicios y Características

Un servicio es un grupo de capacidades. Por ejemplo, un reloj inteligente puede medir su ritmo cardíaco, rastrear su actividad física durante el día y rastrear sus patrones de sueño. Estas tres capacidades, por ejemplo, existirían en un servicio llamado servicio de salud. Al agrupar capacidades en servicios, los dispositivos centrales permiten que los dispositivos periféricos encuentren, seleccionen e interactúen rápidamente con los servicios que desean. Cualquier servicio tiene un código de **identificación único** llamado **UUID**. Este código puede tener una longitud de 16 bits o 32 bits para los servicios de especificación de Bluetooth® oficiales, mientras que los servicios de Bluetooth® no oficiales (los que podemos desarrollar) tienen una longitud de 128 bits, los UUID se pueden crear aleatoriamente. Un perfil es un grupo de servicios.

Dentro de cada servicio existirá una lista de características. Cada una de estas características representa una capacidad única del dispositivo central. En el ejemplo anterior, el servicio de salud tendría tres características (frecuencia cardíaca, actividad física y patrón de sueño). Una vez que el dispositivo periférico descubre estas características, puede escribir información, solicitar información y suscribirse a actualizaciones de estas características. Cualquier característica, como los servicios, tiene un UUID de 16 o 128 bits.

Health Profile (UUID)



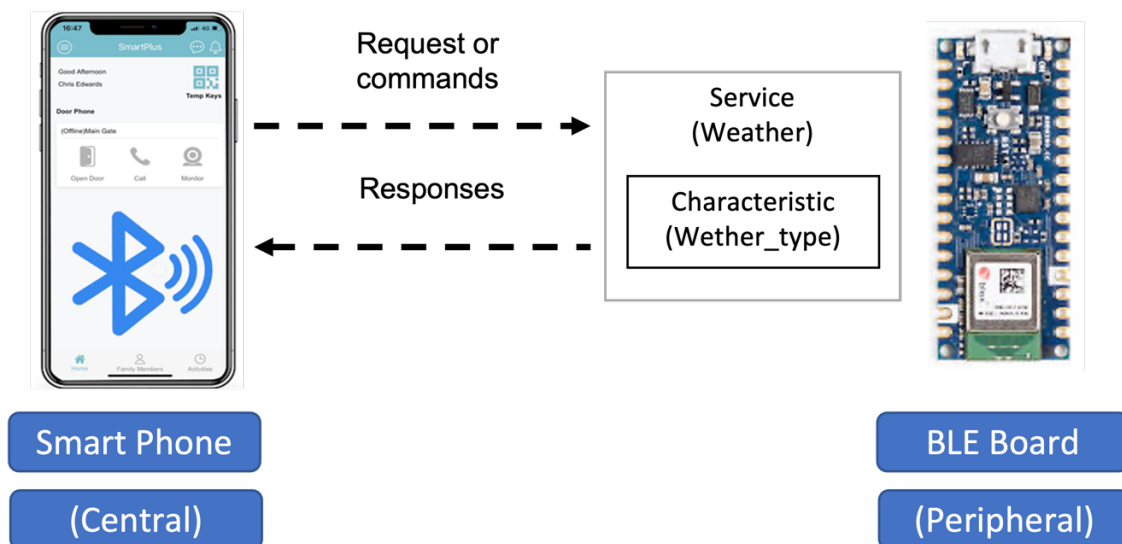
Ejemplo de servicio de salud.

Intercambio de información en Bluetooth® Low Energy

Hay **tres** formas en que se pueden **intercambiar** datos entre dos dispositivos conectados: *leyendo, escribiendo o notificando*.

- La **lectura** ocurre cuando un dispositivo periférico le pide al dispositivo central información específica, piense en un teléfono inteligente que le pide a un reloj inteligente la información de la actividad física, este es un ejemplo de lectura.
- La **escritura** ocurre cuando un dispositivo periférico escribe información específica en el dispositivo central, piensa en un teléfono inteligente cambiando la contraseña de un reloj inteligente, este es un ejemplo de escritura.
- La **notificación** ocurre cuando un dispositivo central ofrece información al dispositivo periférico mediante una notificación, piense en un reloj inteligente que notifica a un teléfono inteligente que su batería está baja y necesita recargarse.

Bueno, eso es lo que necesitamos saber sobre Bluetooth® Low Energy por ahora. Las especificaciones de Bluetooth® son bastante extensas pero interesantes de leer y aprender. Si desea obtener más información sobre Bluetooth® Low Energy, consulte “Primeros pasos con Bluetooth® Low Energy” de Kevin Townsend, Carles Cufí, Akiba y Robert Davidson.



Arduino BLE Ejemplo 1 – Indicador de nivel de batería (Battery Monitor)

En este ejemplo, se explicará cómo se puede leer el nivel de una batería conectada al pin A0 de un Arduino, usando un teléfono inteligente a través de BLE. Se usará el ejemplo proporcionado por la librería **ArduinoBLE.h**.

1. Primero debe instalar la biblioteca ArduinoBLE.h desde el administrador de bibliotecas.
2. Enseguida, vaya a Sketch -> Include library -> Manage Libraries y busque **ArduinoBLE** y simplemente instálelo.

Código de ejemplo:

```
/*  
Battery Monitor  
  
This example creates a Bluetooth® Low Energy peripheral with the standard battery service and  
level characteristic. The A0 pin is used to calculate the battery level.  
  
The circuit:  
- Arduino MKR WiFi 1010, Arduino Uno WiFi Rev2 board, Arduino Nano 33 IoT,  
  Arduino Nano 33 BLE, or Arduino Nano 33 BLE Sense board.  
  
You can use a generic Bluetooth® Low Energy central app, like LightBlue (iOS and Android) or  
nRF Connect (Android), to interact with the services and characteristics  
created in this sketch.  
  
This example code is in the public domain.  
*/  
  
#include <ArduinoBLE.h>  
  
// Bluetooth® Low Energy Battery Service  
BLEService batteryService("180F");  
  
// Bluetooth® Low Energy Battery Level Characteristic  
BLEUnsignedCharCharacteristic batteryLevelChar("2A19", // standard 16-bit characteristic UUID  
  BLERead | BLENotify); // remote clients will be able to get notifications if this characteristic changes  
  
int oldBatteryLevel = 0; // last battery level reading from analog input  
long previousMillis = 0; // last time the battery level was checked, in ms  
  
void setup() {  
  Serial.begin(9600); // initialize serial communication  
  while (!Serial);  
  
  pinMode(LED_BUILTIN, OUTPUT); // initialize the built-in LED pin to indicate when a central is connected  
  
  // begin initialization  
  if (!BLE.begin()) {  
    Serial.println("starting BLE failed!");  
  
    while (1);  
  }  
}
```

```

}

/* Set a local name for the Bluetooth® Low Energy device
   This name will appear in advertising packets
   and can be used by remote devices to identify this Bluetooth® Low Energy device
   The name can be changed but maybe be truncated based on space left in advertisement packet
*/
BLE.setLocalName("BatteryMonitor");
BLE.setAdvertisedService(batteryService); // add the service UUID
batteryService.addCharacteristic(batteryLevelChar); // add the battery level characteristic
BLE.addService(batteryService); // Add the battery service
batteryLevelChar.writeValue(oldBatteryLevel); // set initial value for this characteristic

/* Start advertising Bluetooth® Low Energy. It will start continuously transmitting Bluetooth® Low Energy
   advertising packets and will be visible to remote Bluetooth® Low Energy central devices
   until it receives a new connection */

// start advertising
BLE.advertise();

Serial.println("Bluetooth® device active, waiting for connections...");
}

void loop() {
  // wait for a Bluetooth® Low Energy central
  BLEDevice central = BLE.central();

  // if a central is connected to the peripheral:
  if (central) {
    Serial.print("Connected to central: ");
    // print the central's BT address:
    Serial.println(central.address());
    // turn on the LED to indicate the connection:
    digitalWrite(LED_BUILTIN, HIGH);

    // check the battery level every 200ms
    // while the central is connected:
    while (central.connected()) {
      long currentMillis = millis();
      // if 200ms have passed, check the battery level:
      if (currentMillis - previousMillis >= 200) {
        previousMillis = currentMillis;
        updateBatteryLevel();
      }
    }
    // when the central disconnects, turn off the LED:
    digitalWrite(LED_BUILTIN, LOW);
    Serial.print("Disconnected from central: ");
    Serial.println(central.address());
  }
}

void updateBatteryLevel() {

```

```

/* Read the current voltage level on the A0 analog input pin.
   This is used here to simulate the charge level of a battery.
   */
int battery = analogRead(A0);
int batteryLevel = map(battery, 0, 1023, 0, 100);

if (batteryLevel != oldBatteryLevel) { // if the battery level has changed
  Serial.print("Battery Level % is now: "); // print it
  Serial.println(batteryLevel);
  batteryLevelChar.writeValue(batteryLevel); // and update the battery level characteristic
  oldBatteryLevel = batteryLevel; // save the level for next comparison
}
}

```

Explicación del código de la aplicación “indicador de nivel de batería” Bluetooth de Arduino.

```

#include <ArduinoBLE.h>

// Bluetooth® Low Energy Battery Service
BLEService batteryService("180F");

// Bluetooth® Low Energy Battery Level Characteristic
BLEUnsignedCharCharacteristic batteryLevelChar("2A19", BLERead | BLENotify);

int oldBatteryLevel = 0; // last battery level reading from analog input
long previousMillis = 0; // last time the battery level was checked, in ms

```

La primera línea del código debe incluir el archivo ArduinoBLE.h.

Enseguida declaramos el **Servicio** de batería y las **características** del nivel de batería, en esta parte se darán dos permisos: **BLERead** y **BLENotify**.

BLERead permitirá que los dispositivos centrales (**Smart Phone**) lean datos del dispositivo periférico (**Arduino**).

BLENotify permite que los clientes remotos reciban notificaciones si esta característica cambia.

Se crearan dos variables, oldBatteryLevel de tipo “int” y previousMillis de tipo “long”.

int oldBatteryLevel = Almacenara la lectura del último nivel de batería desde la entrada analógica.

long previousMillis = Almacenara en millis, la última vez que se verificó el nivel de la batería, en ms.

Ahora vamos a saltar a la función de configuración.

```

void setup() {
  Serial.begin(9600);

  pinMode(LED_BUILTIN, OUTPUT);
  if (!BLE.begin()) {
    Serial.println("starting BLE failed!");

    while (1);
  }
}

```

Aquí se inicializará la comunicación serial y el servicio de BLE, el cual hará una espera a que se abra el monitor en serie.

```
BLE.setLocalName("BatteryMonitor");
BLE.setAdvertisedService(batteryService);
batteryService.addCharacteristic(batteryLevelChar);
BLE.addService(batteryService);
batteryLevelChar.writeValue(oldBatteryLevel);
```

Se establece un nombre local para el dispositivo BLE. Este nombre aparecerá en los paquetes publicitarios y puede ser utilizado por dispositivos remotos para identificar este dispositivo BLE.

También se agrega el valor del UUID del servicio y la característica que previamente se declaró usando las variables: `batteryService` y `batteryLevelChar`.

```
BLE.advertise();
Serial.println("Bluetooth device active, waiting for connections...");
}
```

Aquí, se inicia el servicio de anuncios (advertising), el dispositivo comenzará a transmitir continuamente paquetes de publicidad BLE y será visible para los dispositivos centrales BLE remotos hasta que reciba una nueva conexión.

```
void loop() {
  BLEDevice central = BLE.central();

  if (central) {
    Serial.print("Connected to central: ");
    Serial.println(central.address());
    digitalWrite(LED_BUILTIN, HIGH);
  }
}
```

En la función "loop", una vez que todo esté configurado y haya comenzado la publicidad, el dispositivo esperará a cualquier dispositivo central. Una vez que este conectado, mostrará la dirección MAC del dispositivo y encenderá el LED incorporado.

```
while (central.connected()) {

  int battery = analogRead(A0);
  int batteryLevel = map(battery, 0, 1023, 0, 100);
  Serial.print("Battery Level % is now: ");
  Serial.println(batteryLevel);
  batteryLevelChar.writeValue(batteryLevel);
  delay(200);

}
```

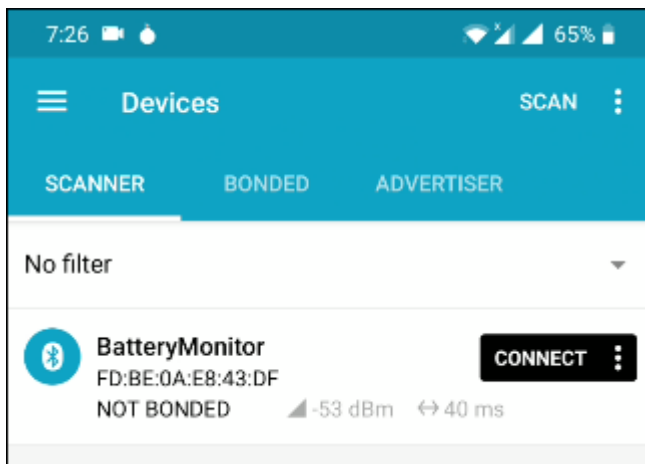
Ahora, comenzará a leer el voltaje analógico del pin "A0", que será un valor entre 0 y 1023 y lo mapeará en el rango de 0 a 100. Imprimirá el nivel de la batería en el monitor serial y el valor se escribirá en la característica de `batteryLevelChar` y esperará 200 ms. Después de eso, todo el bucle se ejecutará nuevamente siempre y cuando que el dispositivo central esté conectado a este dispositivo periférico.


```
}  
digitalWrite(LED_BUILTIN, LOW);  
Serial.print("Disconnected from central: ");  
Serial.println(central.address());  
}
```

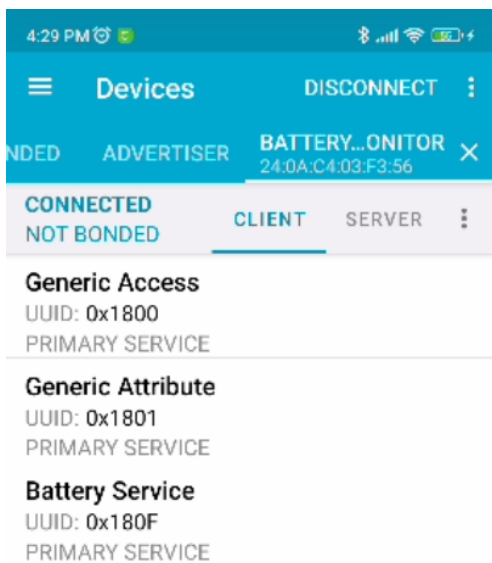
Una vez desconectado, se mostrará un mensaje en el dispositivo central y el LED se apagará.

Instalación de la aplicación para Android

En su teléfono inteligente Android, instale la aplicación "nRF Connect". Ábralo e inicie el escáner. pestaña.



Verá el dispositivo “**BatteryMonitor**” en la lista de dispositivos, ahora toque conectar y se abrirá una nueva.



Vera el servicio “Baterly Service” proceda a tocarlo.

Battery Service

UUID: 0x180F

PRIMARY SERVICE

Battery Level



UUID: 0x2A19



Properties: NOTIFY, READ


Descriptors:

Client Characteristic Configuration



UUID: 0x2902

Observe que se mostrara dos opciones mas, entre ellas Battery level la cual tiene dos modificadores el primero es una flecha hacia abajo  y el segundo tres flechas hacia abajo 

Toque el primero  se mostrara el valor en porcentaje de la carga de la bateria

Battery Service

UUID: 0x180F

PRIMARY SERVICE

Battery Level



UUID: 0x2A19

Properties: NOTIFY, READ

Value: 13%

Descriptors:

Client Characteristic Configuration



UUID: 0x2902

Si toca el el segundo  se activaran las actualizaciones de dicho valor.

Battery Service

UUID: 0x180F

PRIMARY SERVICE

Battery Level



UUID: 0x2A19

Properties: NOTIFY, READ

Value: 11%

Descriptors:

Client Characteristic Configuration



UUID: 0x2902

Value: Notifications enabled

Arduino BLE Ejemplo 2 – Encender y apagar un Led

En este ejemplo, se usará la característica BLEWrite para controlar el estado de un Led conectado a una tarjeta Arduino, y mediante un teléfono inteligente a través de BLE, encenderlo y apagarlo. Se usará el ejemplo proporcionado por la librería **ArduinoBLE.h**.

Código de ejemplo explicado paso a paso del segundo ejemplo “Led BLE”.

```
// La primera línea del código debe incluir la biblioteca ArduinoBLE.h.
#include <ArduinoBLE.h>

// Enseguida declararemos el Servicio “ledService”
BLEService ledService("19B10000-E8F2-537E-4F6C-D104768A1214"); // Bluetooth® Low Energy LED Service

// Se declara la característica de “switchCharacteristic”, con los permisos: BLERead y BLEWrite.
BLEByteCharacteristic switchCharacteristic("19B10001-E8F2-537E-4F6C-D104768A1214", BLERead | BLEWrite);

// BLERead permite que los dispositivos centrales (Smart Phone) lean datos del dispositivo periférico (Arduino).
// BLEWrite permite que los dispositivos centrales (Smart Phone) envíen nuevos datos (write).

// Se crea la variable entera (int) ledPin, donde se almacenará el pin GPIO, donde esté conectado el Led.
const int ledPin = 32;

// Se inicializa la comunicación serial.
void setup() {
  Serial.begin(9600);
  while (!Serial);

  // Declaramos el pin asignado al Led como salida.
  pinMode(ledPin, OUTPUT);

  // Se verifica que el servicio de BLE esté listo y en espera.
  if (!BLE.begin()) {
    Serial.println("starting Bluetooth® Low Energy module failed!");

    while (1);
  }

  // Se configura el nombre y el UUID del servicio.
  BLE.setLocalName("BLE-Led");
  BLE.setAdvertisedService(ledService);

  // Se agrega la característica al servicio.
  ledService.addCharacteristic(switchCharacteristic);

  // Se agrega el servicio.
  BLE.addService(ledService);

  // Se configura el valor inicial de la característica.
  switchCharacteristic.writeValue(0);
```

```

// Se inicia el anuncio del servicio.

BLE.advertise();

Serial.println("BLE LED Peripheral");
}

void loop() {
// Se inicia la espera de conexiones de dispositivos centrales BLE.
BLEDevice central = BLE.central();

// Si un dispositivo central se conecta.
if (central) {
  Serial.print("Connected to central: ");
  // Se imprime la dirección MAC del dispositivo central.
  Serial.println(central.address());

  // Mientras el dispositivo central esté conectado al periférico:
  while (central.connected()) {
    // Si el dispositivo remoto escribe un nuevo valor en la característica,
    // se usa el nuevo valor para controlar el LED:
    if (switchCharacteristic.written()) {
      if (switchCharacteristic.value()) { // Cualquier valor diferente de 0.
        Serial.println("LED on");
        digitalWrite(ledPin, HIGH); // El Led se enciende.
      } else { // Si el valor es 0
        Serial.println(F("LED off"));
        digitalWrite(ledPin, LOW); // El Led se apaga.
      }
    }
  }
}

// Cuando el dispositivo central se desconecte, se imprime el mensaje:
Serial.print(F("Disconnected from central: "));
Serial.println(central.address());
}
}

```

Arduino BLE Ejemplo 3 – Aplicación de Clima

En este ejemplo, crearemos una aplicación BLE basada en Arduino que tome los valores de temperatura y humedad de un sensor DHT11 y los pasaremos a un servicio con dos características, posteriormente se creará una aplicación móvil en App Inventor con la cual podremos conectarnos al dispositivo BLE para mostrar en la pantalla del teléfono móvil los valores del clima actual.

Código Arduino para el ESP32 Dev Module

```
/*  
  Weather Monitor  
  
  This example creates a Bluetooth® Low Energy peripheral with the weather service and  
  level characteristic.  
  
  This example code is in the public domain.  
  
  Written by Gustavo Reynaga @gsreynaga.  
  
  For: ESP32 Dev C (38 pins)  
  
  The Service and characteristic UUID are generated by https://www.uuidgenerator.net/  
  
  Weather service:      14cae221-7236-4804-8fc5-4c32a1d65e67  
  Temperature characteristic: 14cae222-7236-4804-8fc5-4c32a1d65e67  
  Humidity characteristic:  14cae223-7236-4804-8fc5-4c32a1d65e67  
  
*/  
  
//Set BLE library  
  
#include <ArduinoBLE.h>  
  
#include "DHT.h"  
#define DHTPIN 33  
#define DHTTYPE DHT11  
  
int pinLed =32;  
  
// Bluetooth® Low Energy Weather Service  
BLEService TEMPHUMI("19B10010-E8F2-537E-4F6C-D104768A1214"); // BLE Service Temperature & Humidity  
  
// BLE Temperature & Humidity Characteristic - custom 128-bit UUID, read by central  
//BLEFloatCharacteristic  
BLEIntCharacteristic DHT11_TEMP( "19B10011-E8F2-537E-4F6C-D104768A1214", BLERead | BLENotify);  
BLEIntCharacteristic DHT11_HUMI( "19B10012-E8F2-537E-4F6C-D104768A1214", BLERead | BLENotify);  
  
long previousMillis = 0; // will store last time "Weather data" was updated  
  
//float send_data_temp;  
//float send_data_humi;  
int send_data_temp;  
int send_data_humi;  
  
// DHT11  
DHT dht(DHTPIN, DHTTYPE);  
  
void setup() {  
  Serial.begin(9600); // initialize serial communication
```

```

pinMode(pinLed, OUTPUT);
// begin initialization
if (!BLE.begin()) {
  Serial.println("starting BLE failed!");

  while (1);
}
dht.begin();

// set advertised local name and service UUID:
BLE.setLocalName("32Dev_Weather");
BLE.setAdvertisedService(TEMPHUMI);
// add the characteristic to the service
TEMPHUMI.addCharacteristic(DHT11_TEMP);
TEMPHUMI.addCharacteristic(DHT11_HUMI);
// add service
BLE.addService(TEMPHUMI);

// begin advertising BLE service:
BLE.advertise();

Serial.println("BLE service start.");
}

void loop() {
  // listen for BLE peripherals to connect:
  BLEDevice central = BLE.central();

  // if a central is connected to peripheral:
  if (central) {
    Serial.print("Connected... ");
    digitalWrite(pinLed, HIGH);
    // print the central's MAC address:
    Serial.println(central.address());

    // while the central is still connected to peripheral:
    while (central.connected()) {
      long currentMillis = millis();
      // if 5000ms have passed, check the Weather sensor:
      if (currentMillis - previousMillis >= 5000) {
        // save the last time "Weather data" was updated
        previousMillis = currentMillis;
        updateTemp();
      }
    }
  }
  // when the central disconnects, print it out:
  Serial.print(F("Disconnected from central: "));
  digitalWrite(pinLed, LOW);
  Serial.println(central.address());
}

```

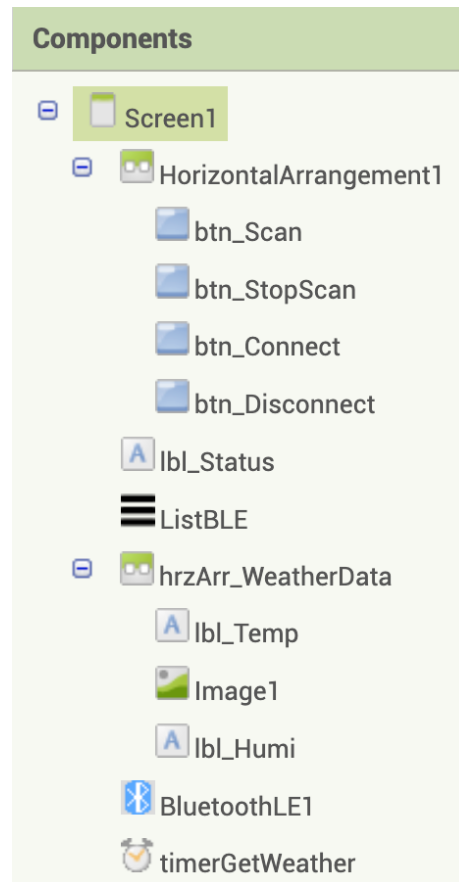
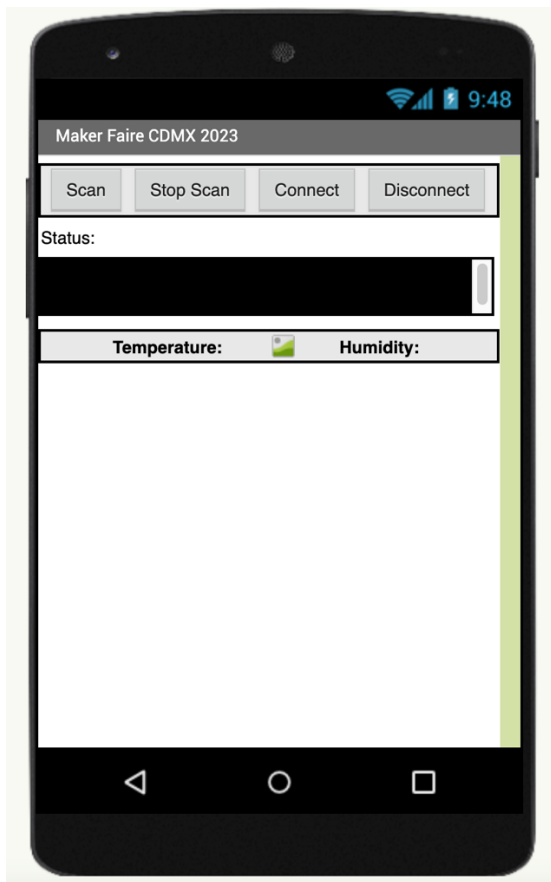
```
}

void updateTemp(){
  // Read temperature as Celsius (the default) & Humidity
  float h = dht.readHumidity();
  float t = dht.readTemperature();
  // Check if any reads failed and exit early (to try again).
  if (isnan(h) || isnan(t)) {
    Serial.println(F("Failed to read from DHT sensor!"));
    return;
  }
  int temp = round(t);
  send_data_temp = temp; //send temperature
  DHT11_TEMP.setValue(send_data_temp);
  int humi = round(h);
  send_data_humi = humi; //send humidity
  DHT11_HUMI.setValue(send_data_humi);

  //send Weather data to serial monitor (only for debug)
  Serial.print("Temperature: ");
  Serial.print(t);
  Serial.println(F("°C "));
  Serial.print("Humidity: ");
  Serial.print(h);
  Serial.println(F("% "));
}
```

Código App Inventor

Diseñador



Bloques

initialize global serviceTEMPHUMI to "19B10010-E8F2-537E-4F6C-D104768A1214"

initialize global characteristicDHT11_TEMP to "19B10011-E8F2-537E-4F6C-D104768A1214"

initialize global characteristicDHT11_HUMI to "19B10012-E8F2-537E-4F6C-D104768A1214"

to callHumi
do
call BluetoothLE1 .ReadShorts
serviceUuid get global serviceTEMPHUMI
characteristicUuid get global characteristicDHT11_HUMI
signed false
call BluetoothLE1 .RegisterForShorts
serviceUuid get global serviceTEMPHUMI
characteristicUuid get global characteristicDHT11_HUMI
signed false
end

to callTemp
do
call BluetoothLE1 .ReadShorts
serviceUuid get global serviceTEMPHUMI
characteristicUuid get global characteristicDHT11_TEMP
signed false
call BluetoothLE1 .RegisterForShorts
serviceUuid get global serviceTEMPHUMI
characteristicUuid get global characteristicDHT11_TEMP
signed false
end

when BluetoothLE1 .DeviceFound
do
set ListBLE .ElementsFromString to BluetoothLE1 .DeviceList
end

when BluetoothLE1 .Connected
do
set lbl_Status .Text to "Status: Connected"
set ListBLE .Visible to false
end

when BluetoothLE1 .Disconnected
do
set lbl_Status .Text to "Status: Disconnected"
set lbl_Temp .Text to "Temperature: "
set lbl_Humi .Text to "Humidity: "
end

when BluetoothLE1 .ShortsReceived
serviceUuid
characteristicUuid shortValues
do
if upcase get characteristicUuid == get global characteristicDHT11_TEMP
then
set lbl_Temp .Text to join "Temperature: "
select list item list get shortValues
index 1
"°C"
end
if upcase get characteristicUuid == get global characteristicDHT11_HUMI
then
set lbl_Humi .Text to join "Humidity: "
select list item list get shortValues
index 1
"%"
end
end

when btn_Connect .Click
do
call BluetoothLE1 .Connect
index ListBLE .SelectionIndex
set lbl_Status .Text to "Status: Connecting..."
end

when btn_Disconnect .Click
do
call BluetoothLE1 .Disconnect
end

when btn_Scan .Click
do
call BluetoothLE1 .StartScanning
set lbl_Status .Text to "Status: Scanning..."
set ListBLE .Visible to true
end

when btn_StopScan .Click
do
call BluetoothLE1 .StopScanning
set lbl_Status .Text to "Status: Stopped Scanning"
end

when timerGetWeather .Timer
do
if BluetoothLE1 .IsDeviceConnected
then
call callTemp
call callHumi
end
end

when Screen1 .Initialize
do
set timerGetWeather .TimerEnabled to true
end

Fuentes:

<https://docs.arduino.cc/tutorials/nano-33-ble-sense/ble-device-to-device>
<https://www.arduino.cc/reference/en/libraries/arduino-ble/>
<https://docs.arduino.cc/tutorials/nano-33-ble-sense/ble-device-to-device>
<https://ladvien.com/arduino-nano-33-bluetooth-low-energy-setup/>
<https://rootsaid.com/arduino-ble-example/?amp>

Regresar a la Presentación.