ChatGPT

ArduinoBLE Library – API Documentation

Version: ArduinoBLE v1.4.0 (Arduino Bluetooth® Low Energy library)
Description: The ArduinoBLE library enables Bluetooth® Low Energy (BLE) connectivity on supported Arduino boards (e.g. Nano 33 BLE, Nano 33 IoT, MKR WiFi 1010, UNO R4 WiFi, etc.). It supports both *Peripheral* mode (where the Arduino advertises services and characteristics) and *Central* mode (where the Arduino scans for and connects to peripherals). This document compiles the official API reference for the main classes in ArduinoBLE:

- BLE – The BLE subsystem control class (for initializing BLE, scanning, advertising, etc.).
- BLEDevice – Represents a remote BLE device (peripheral when acting as central, or a central when acting as peripheral) and provides information and actions for connections or discovered devices.
- BLEService – Represents a BLE GATT service, either one provided by the local device or discovered on a remote device. Services contain characteristics.
- BLECharacteristic – Represents a BLE GATT characteristic, either one provided by the local device or discovered on a remote device. Characteristics hold a value and may have associated descriptors.
- BLEDescriptor – Represents a BLE GATT descriptor, an attribute that provides additional information about a characteristic (such as description or configuration).

Each section below details the class API: function signatures, descriptions, parameters, return values, and usage examples. Code snippets are in C++ for Arduino.

Table of Contents:

- hasLocalName()
- deviceName()
- appearance()
- rssi()
- advertisedServiceUuidCount()
- advertisedServiceUuid()
- hasAdvertisedServiceUuid()
- hasService()
- hasCharacteristic()
- serviceCount()
- service()
- characteristicCount()
- characteristic()
- connect()
- disconnect()
- discoverAttributes()
- poll()
- connected()
- BLEService Class
- BLEService() Constructor
- uuid()
- addCharacteristic()
- characteristic()
- characteristicCount()
- Operator bool
- BLECharacteristic Class
- BLECharacteristic() Constructors
- uuid()
- properties()
- value()
- valueLength()
- valueSize()
- readValue()
- writeValue()
- setEventHandler()
- subscribe() / unsubscribe()
- subscribed()
- canRead() / canWrite()
- canSubscribe() / canUnsubscribe()
- hasDescriptor()
- descriptorCount()
- descriptor()
- written()
- valueUpdated()
- addDescriptor()
- Operator bool
- BLEDescriptor Class
- BLEDescriptor() Constructors

BLE Class

The **BLE** class is a globally accessible object (instantiated as `BLE`) used to initialize and manage the Bluetooth® Low Energy module on the Arduino board. It provides methods to begin or end BLE functionality, set device advertising parameters (name, advertised services), start/stop advertising, perform scanning for peripherals (as a central), and handle connections. Most BLE operations begin with calling `BLE.begin()` and then using other BLE methods for the desired mode (peripheral or central).

> **Note:** The `BLE` object is used for high-level control of the BLE radio. Many of its methods are static-like in usage. The `BLE` class is not explicitly constructed by the user (you use the provided global instance). All functions are called as `BLE.functionName()`.

BLE.begin()

Description: Initializes the Bluetooth® Low Energy device and begins BLE functionality. This must be called first before using other BLE functions. In peripheral mode, this prepares the device for advertising (services should be added before advertising). In central mode, this prepares the device for scanning.

Syntax:

```
int BLE.begin();
```

Parameters: None.

Returns: `1` on success (BLE module initialized successfully), or `0` on failure (e.g., if the BLE hardware could not be started).

Example:

```
if (!BLE.begin()) {
  Serial.println("starting Bluetooth® Low Energy module failed!");
  while (1);
}
```

In this example, the sketch attempts to start the BLE module and enters an infinite loop if initialization fails.

BLE.end()

Description: Shuts down the BLE module and releases any used resources. Call this to turn off BLE functionality (e.g., to save power or reset the BLE state).

Syntax:

```
void BLE.end();
```

Parameters: None.

Returns: Nothing.

Example: After calling `BLE.end()`, the device will stop advertising or scanning and any active connections will terminate.

BLE.setDeviceName()

Description: Sets the Bluetooth GAP **Device Name** attribute of the local device. The device name is a user-friendly name stored in the Generic Access service (separate from the advertised local name). If not set, it defaults to `"Arduino"`.

Syntax:

```
void BLE.setDeviceName(const char* name);
```

Parameters: - `name`: The desired device name (null-terminated string). Maximum length is 20 characters (longer names will be truncated by the BLE stack).

Returns: Nothing.

Example:

```
BLE.setDeviceName("MyArduino");
```

This sets the GAP device name to "MyArduino". This name may be visible to peers querying the Generic Access service after connecting.

BLE.setLocalName()

Description: Sets the local name used in advertising packets for the device. This is the name that other BLE devices can see when they scan for peripherals. If not set, it defaults to `"Arduino"`.

Syntax:

```
void BLE.setLocalName(const char* name);
```

Parameters: - `name`: The local name to advertise (null-terminated string). Typically up to Ble's maximum advertising name length (often 31 bytes including BLE data overhead).

Returns: Nothing.

Example:

```
BLE.setLocalName("EnvSensor");
// Now the device will advertise with the name "EnvSensor"
```

A central scanning for devices by name could look for "EnvSensor".

BLE.setAdvertisedService() / BLE.setAdvertisedServiceUuid()

Description: Configures which service UUID(s) to include in the advertisement data, so that other devices scanning can identify supported services before connecting. You can pass a `BLEService` object or a UUID string. This should be called **after** creating and configuring the service(s) and before calling `BLE.advertise()`.

Syntax:

```
void BLE.setAdvertisedService(BLEService& service);
void BLE.setAdvertisedServiceUuid(const char* uuid);
```

Parameters: - `service`: A `BLEService` object that has been created (and typically had characteristics added). The service's UUID will be included in advertising data. - `uuid`: A 16-bit or 128-bit UUID string to advertise (e.g., `"180D"` or `"19B10000-E8F2-537E-4F6C-D104768A1214"`).

Returns: Nothing.

Example:

```
BLEService tempService("1809");           // 0x1809 = Health Thermometer
service UUID
BLE.setAdvertisedService(tempService);    // Advertise this service's UUID
BLE.addService(tempService);
// ... add characteristics to tempService ...
BLE.advertise();
```

In this example, the device will advertise the Health Thermometer service UUID in its BLE advertising packets. Scanning devices can filter by this UUID to find this peripheral.

BLE.addService()

Description: Registers a BLE service with the local BLE device, making it available to remote centrals. All characteristics should be added to the service *before* adding the service to BLE. After adding the service, you can start advertising it.

Syntax:

```
bool BLE.addService(BLEService& service);
```

Parameters: - `service`: The `BLEService` object to add (which should have any desired characteristics already added to it).

Returns: `true` on success (service added and ready), or `false` on failure.

Example:

```
BLEService ledService("19B10000-E8F2-537E-4F6C-D104768A1214");
BLE.addService(ledService);
```

After adding the service, calling `BLE.advertise()` will include the service's UUID in advertising (if configured via `BLE.setAdvertisedService()` as shown above).

BLE.advertise()

Description: Starts advertising the BLE peripheral. Once called (after `BLE.begin()` and setting up services/characteristics), the device will broadcast advertising packets and be discoverable by centrals. Advertising will continue until `BLE.stopAdvertise()` is called or a connection is established (in which case advertising may stop automatically).

Syntax:

```
int BLE.advertise();
```

Parameters: None.

Returns: `1` on success (advertising started), or `0` on failure.

Example:

```
// Begin BLE and set up service/characteristics...
BLE.advertise();
Serial.println("BLE advertising started!");
```

This will make the device visible to others. Typically you set the local name and advertised service before calling this.

BLE.stopAdvertise()

Description: Stops advertising. If the device was advertising, it will no longer be discoverable by other devices.

Syntax:

```
void BLE.stopAdvertise();
```

Parameters: None.

Returns: Nothing.

Example: After calling `BLE.stopAdvertise()`, a previously advertising peripheral will cease broadcasting its presence (useful to conserve power or if you only want to advertise for a certain time/window).

BLE.scan()

Description: Instructs the local device (when in Central role) to start scanning for nearby BLE peripherals. This will actively scan for any advertising BLE devices. Typically you call `BLE.scan()` and then repeatedly call `BLE.available()` to retrieve discovered devices. Scanning continues until you stop it or possibly until a timeout if one is set using other scan functions. This overload scans for all devices unfiltered.

Syntax:

```
bool BLE.scan();
```

Parameters: None.

Returns: `true` if scanning successfully started, `false` if it failed to start.

Example:

```
BLE.scan();
Serial.println("Scanning for BLE devices...");
```

After this, you can use `BLE.available()` to get discovered devices as they are found.

BLE.scanForName() / BLE.scanForAddress() / BLE.scanForUuid()

Description: These functions start scanning with a filter for a specific device name, address, or service UUID. Using a filter can make discovery faster by only reporting devices that match the criteria. While scanning, you still retrieve devices with `BLE.available()`, but only devices matching the filter will be returned.

Syntax:

```
bool BLE.scanForName(const char* name);
bool BLE.scanForName(const char* name, bool stopOnMatch);
bool BLE.scanForAddress(const char* address);
bool BLE.scanForAddress(const char* address, bool stopOnMatch);
bool BLE.scanForUuid(const char* uuid);
bool BLE.scanForUuid(const char* uuid, bool stopOnMatch);
```

Parameters: - `name`: The advertised local name to look for (exact match). - `address`: The MAC address (in string form, typically `"XX:XX:XX:XX:XX:XX"`) to look for. - `uuid`: The service UUID to look for (as a string, e.g., `"180D"` or full 128-bit). - `stopOnMatch`: *(Optional)* If `true`, the scanning will stop as soon as a device matching the filter is found. If `false` or not provided, scanning continues until manually stopped or a timeout elapses.

Returns: `true` if scanning started successfully (with the filter applied), or `false` on failure.

Example:

```
BLE.scanForName("EnvSensor");
// Only devices advertising the name "EnvSensor" will be reported via
BLE.available()
```

Similarly, `BLE.scanForUuid("180D")` would scan for devices advertising the Heart Rate service UUID 0x180D.

BLE.available()

Description: Checks if a new BLE peripheral device has been discovered (when scanning as a central), and returns a `BLEDevice` object for it. Each call to `BLE.available()` yields the next discovered device from the scan queue. Use this in a loop after starting a scan to handle found devices.

Syntax:

```
BLEDevice BLE.available();
```

Parameters: None.

Returns: A `BLEDevice` representing a discovered peripheral. If no new device is available, returns a **falsey**/invalid BLEDevice (which evaluates to `false` in boolean context).

Example:

```
BLE.scan();
while (true) {
  BLEDevice peripheral = BLE.available();
  if (peripheral) {
    // A device was found
    Serial.print("Found device: ");
    Serial.println(peripheral.address());
    break;
  }
}
```

In this snippet, the code scans and breaks out when the first device is found, printing its address.

BLE.stopScan()

Description: Stops an ongoing scan for BLE devices. After calling this, no new devices will be discovered until scanning is started again.

Syntax:

```
void BLE.stopScan();
```

Parameters: None.

Returns: Nothing.

Example: If you began scanning with `BLE.scan()` or `BLE.scanFor...()`, you can call `BLE.stopScan()` to halt the process (for example, after a certain time or after finding a desired device).

BLE.central()

Description: When the local device is in Peripheral mode, this function checks for and returns a `BLEDevice` object representing a connected central device. It is typically used to detect when a central connects or disconnects. `BLE.central()` will return an object evaluating to true when a central is connected.

Syntax:

```
BLEDevice BLE.central();
```

Parameters: None.

Returns: If a central device is currently connected to this peripheral, a `BLEDevice` representing the central is returned (evaluates to true). If no central is connected, an invalid BLEDevice (evaluates to false) is returned.

Example:

```
BLEDevice central = BLE.central();
if (central) {
  // A central is connected
  Serial.print("Connected to central: ");
  Serial.println(central.address());
}
```

This is often used in a loop to wait for a connection. Once `BLE.central()` returns a valid device, you know a central has connected. You might then proceed to communicate with it or just monitor its connection.

BLE.connected()

Description: Checks if the local BLE device is currently connected to another BLE device. This is a convenient way to query connection status (either as a peripheral with a central connected, or as a central connected to a peripheral).

Syntax:

```
bool BLE.connected();
```

Parameters: None.

Returns: `true` if at least one BLE connection is active (the board is connected to a peer device), `false` if not connected.

Example:

```
if (BLE.connected()) {
  Serial.println("Device is connected to a peer.");
}
```

This could be used to guard code that should only run when a connection exists.

BLE.disconnect()

Description: Forces a disconnect of the current BLE connection. If the device is connected to a central (in peripheral mode) or to a peripheral (in central mode), this will terminate that connection.

Syntax:

```
bool BLE.disconnect();
```

Parameters: None.

Returns: `true` if a connection was present and the disconnect command was issued, or `false` if there was no connection to disconnect.

Example:

```
if (!BLE.connected()) {
  // no one is connected, perhaps we want to stop advertising
```

```
  BLE.stopAdvertise();
} else {
  // if someone is connected and we need to disconnect them:
  BLE.disconnect();
}
```

After `BLE.disconnect()`, the remote device will be disconnected (and will get a disconnect event on its side).

BLE.address()

Description: Retrieves the Bluetooth address (MAC address) of the local BLE radio. This is the unique 48-bit hardware address of the Arduino's BLE module.

Syntax:

```
String BLE.address();
```

Parameters: None.

Returns: A `String` representing the MAC address in the form `"XX:XX:XX:XX:XX:XX"`.

Example:

```
Serial.print("My BLE address: ");
Serial.println(BLE.address());
```

This might print something like `My BLE address: A4:C1:38:12:34:56`.

BLE.rssi()

Description: Queries the RSSI (Received Signal Strength Indicator) of the currently connected BLE device. This provides the signal strength of the link. It is only valid when a connection is active (either a central connected to us, or us connected to a peripheral).

Syntax:

```
int BLE.rssi();
```

Parameters: None.

Returns: The RSSI value (in dBm) of the peer device in the active connection. Returns 0 if no connection is present.

Example:

```
if (BLE.connected()) {
  int signal = BLE.rssi();
  Serial.print("Connection RSSI: ");
  Serial.print(signal);
  Serial.println(" dBm");
}
```

A higher (less negative) RSSI indicates a stronger signal.

BLE.setAdvertisingInterval()

Description: Sets the advertising interval for BLE advertising packets. The advertising interval controls how frequently the device sends out advertising beacons when `BLE.advertise()` is active. By default, it is 100 ms.

Syntax:

```
void BLE.setAdvertisingInterval(uint16_t advertisingInterval);
```

Parameters: - `advertisingInterval`: The desired advertising interval in units of 0.625 ms. For example, an interval of 160 corresponds to 100 ms (160 * 0.625 ms). The minimum and maximum allowed values depend on BLE specifications (typically between 0x0020 (20 ms) and 0x4000).

Returns: Nothing.

Example:

```
BLE.setAdvertisingInterval(320); // 320 * 0.625ms = 200ms interval
BLE.advertise();
```

This sets the advertising interval to 200 ms, meaning the device advertises roughly 5 times per second.

BLE.setConnectionInterval()

Description: Sets the desired minimum and maximum connection interval for BLE connections. This influences the periodic communication frequency between connected BLE devices. A shorter interval means more frequent connection events (higher throughput, higher power consumption), while a longer interval means less frequent communication (lower power usage).

Syntax:

```
void BLE.setConnectionInterval(uint16_t minimum, uint16_t maximum);
```

Parameters: - `minimum`: The minimum connection interval in units of 1.25 ms. - `maximum`: The maximum connection interval in units of 1.25 ms.

Returns: Nothing.

Example:

```
// Request a connection interval between 10ms and 20ms
BLE.setConnectionInterval(8, 16);  // 8 * 1.25ms = 10ms, 16 * 1.25ms = 20ms
```

This hints to the BLE controller to use a fast connection interval (if supported by the peer). The actual interval will be negotiated with the central device.

BLE.setConnectable()

Description: Controls whether the device allows connections while advertising. By default, `BLE.advertise()` makes the device connectable. If you set the device to non-connectable, it will advertise (broadcast) but remote centrals will not be able to initiate a connection. Non-connectable advertising is useful for beacon scenarios.

Syntax:

```
void BLE.setConnectable(bool connectable);
```

Parameters: - `connectable`: Pass `true` to make the advertising connectable (default), or `false` to advertise in non-connectable mode.

Returns: Nothing.

Example:

```
BLE.setLocalName("BeaconDemo");
BLE.setAdvertisedServiceUuid("180F"); // Battery Service UUID for demonstration
BLE.setConnectable(false);
BLE.advertise();
```

In this example, the device advertises Battery Service data but cannot be connected to, effectively functioning as a beacon.

BLEDevice Class

The **BLEDevice** class represents a remote BLE device. Depending on context, it can represent: - A BLE peripheral that was discovered during a scan (when the Arduino is in Central mode). - A BLE peripheral that the Arduino (in Central mode) has connected to. - A BLE central that has connected to the Arduino (when the Arduino is in Peripheral mode).

BLEDevice provides information such as the device's address, advertised data (name, service UUIDs, appearance, RSSI), and methods to connect or disconnect (for centrals connecting to peripherals). When connected, a BLEDevice can also be used to discover services/characteristics on the peer and read/write them.

You typically obtain a BLEDevice in one of two ways: - As a *scanned device* by calling `BLE.available()` after `BLE.scan()`. - As a *connected device* by calling `BLE.central()` (for a connected central) or as the return value of `BLEDevice.connect()` when connecting to a peripheral.

BLEDevice objects can be checked in boolean context to see if they represent a valid device (e.g., `if (device) {...}` is true if the device is valid or connected).

BLEDevice – General Usage and Properties

A BLEDevice includes both advertising-phase data and connection-phase data: - **Advertising data** (available even before connection): address, advertised local name, advertised service UUIDs, appearance, and RSSI. These are populated if the BLEDevice comes from scanning. - **Connection**: Once connected (by calling `device.connect()` as a central, or when obtained via `BLE.central()` as a peripheral), you can perform service and characteristic discovery on the BLEDevice, and then interact with those services/characteristics.

Important: To use a BLEDevice obtained from scanning, you must call `device.connect()` to establish a connection before you can read characteristics or descriptors. After connecting (and optionally discovering attributes with `discoverAttributes()`), you can use methods like `device.service(...)` or `device.characteristic(...)` to get specific remote services or characteristics. If you already know the UUID of a desired service or characteristic, you can use the provided methods to get them directly.

Operator bool: You can test a BLEDevice object in an `if` or logical context. It will evaluate to `true` if it is a valid device reference (e.g., a scanned device entry or an active connection), and `false` if it is not valid. For example:

```
BLEDevice dev = BLE.available();
if (dev) {
```

```
    // dev is a valid discovered device
}
```

Now we detail the BLEDevice class methods:

BLEDevice.address()

Description: Returns the BLE MAC address of the remote device.

Syntax:

```
String BLEDevice::address();
```

Parameters: None.

Returns: A `String` containing the device's address in hex format (`"XX:XX:XX:XX:XX:XX"`). If the address is not known, an empty string may be returned (address is always known for scanned devices).

Example:

```
BLEDevice dev = BLE.available();
if (dev) {
  Serial.print("Found device at ");
  Serial.println(dev.address());
}
```

This prints the address of a discovered device.

BLEDevice.localName()

Description: Gets the advertised local name of the remote device, if it was advertising one.

Syntax:

```
String BLEDevice::localName();
```

Parameters: None.

Returns: A `String` of the advertised local name of the device. If the device did not advertise a name, this will be an empty string.

Example:

```
if (dev.hasLocalName()) {
  Serial.print("Device name: ");
  Serial.println(dev.localName());
}
```

This retrieves and prints the device's name (if available in the advertisement).

BLEDevice.hasLocalName()

Description: Checks if the remote device's advertisement included a local name.

Syntax:

```
bool BLEDevice::hasLocalName();
```

Parameters: None.

Returns: `true` if a local name was present in the advertising data, `false` if not.

Example: (See example under `localName()` above, where `hasLocalName()` is used to guard access to the name.)

BLEDevice.deviceName()

Description: Retrieves the GAP Device Name of the remote device (from the Generic Access service on the peer). This usually requires that the Arduino has connected to the device and performed service discovery, because the Device Name is a characteristic in the Generic Access service.

Syntax:

```
String BLEDevice::deviceName();
```

Parameters: None.

Returns: A `String` with the device's GAP device name. If not available (e.g., not discovered), returns an empty string.

Usage: Typically, call `device.discoverAttributes()` after connecting, then use `deviceName()`. Not all devices have a device name set or accessible.

Example:

```
BLEDevice peripheral = BLE.available();
if (peripheral) {
  peripheral.connect();
  peripheral.discoverAttributes();
  Serial.println(peripheral.deviceName());
}
```

This connects to a discovered device and prints its GAP device name (if provided by the device).

BLEDevice.appearance()

Description: Returns the Appearance value from the remote device's advertising data or Generic Access service. The appearance is a 16-bit hint of the device's type (for example, generic tag, heart rate sensor, etc., as defined by BLE spec).

Syntax:

```
unsigned short BLEDevice::appearance();
```

Parameters: None.

Returns: A 16-bit unsigned integer representing the appearance category of the device. If unavailable, it may return 0 or a default value.

Example:

```
uint16_t app = dev.appearance();
Serial.print("Appearance code: 0x");
Serial.println(app, HEX);
```

You might decode this value based on BLE specifications (e.g., 0x0040 might indicate a heart rate sensor).

BLEDevice.rssi()

Description: Gets the Received Signal Strength Indicator (RSSI) for the remote device's advertisement. This indicates how strong the signal was when the device was discovered. (This is not updated after connection; for connection RSSI, use `BLE.rssi()` on the central side.)

Syntax:

```
int BLEDevice::rssi();
```

Parameters: None.

Returns: The RSSI value (in dBm) measured during the discovery of the device. Typical values are negative (e.g., -40 is strong, -90 is weak). Returns 0 if not available.

Example:

```
Serial.print("Signal strength: ");
Serial.print(dev.rssi());
Serial.println(" dBm");
```

This prints the RSSI of a scanned device's advertisement.

BLEDevice.advertisedServiceUuidCount()

Description: Returns the number of service UUIDs advertised by the remote device. Some peripherals include one or more service UUIDs in their advertising packets to let centrals know what services are available without a full connection.

Syntax:

```
int BLEDevice::advertisedServiceUuidCount();
```

Parameters: None.

Returns: The count of service UUIDs present in the advertising data. This could be 0 if none were advertised.

Example:

```
int svcCount = dev.advertisedServiceUuidCount();
Serial.print("Advertised service count: ");
Serial.println(svcCount);
```

You can iterate through the indices from `0` to `svcCount - 1` and call `advertisedServiceUuid(index)` to get each one.

BLEDevice.advertisedServiceUuid()

Description: Gets one of the advertised service UUIDs from the remote device's advertising data, by index.

Syntax:

```
String BLEDevice::advertisedServiceUuid(int index);
```

Parameters: - `index`: The index (0-based) of the service UUID to retrieve. Must be less than `advertisedServiceUuidCount()`.

Returns: A `String` containing the UUID at that index, in the standard format (e.g., `"180D"` or full 128-bit string). Returns an empty string if the index is out of range.

Example:

```
for (int i = 0; i < dev.advertisedServiceUuidCount(); i++) {
  Serial.print("Service UUID[");
  Serial.print(i);
  Serial.print("]: ");
  Serial.println(dev.advertisedServiceUuid(i));
}
```

This loops through all advertised service UUIDs and prints them.

BLEDevice.hasAdvertisedServiceUuid()

Description: Checks if a specific service UUID is present in the device's advertising data.

Syntax:

```
bool BLEDevice::hasAdvertisedServiceUuid(const char* uuid);
```

Parameters: - `uuid`: The service UUID to check for (as a string, can be 16-bit or 128-bit format).

Returns: `true` if the UUID was advertised by the device, `false` if not.

Example:

```
if (dev.hasAdvertisedServiceUuid("180F")) {
  Serial.println("Device advertises Battery Service.");
}
```

This would detect if the peripheral advertised the standard Battery Service (UUID 0x180F).

BLEDevice.hasService()

Description: Checks if the remote device (when connected) contains a given service. This requires that services have been discovered (either via advertisement or by performing a service discovery after connecting).

Syntax:

```
bool BLEDevice::hasService(const char* uuid);
```

Parameters: - `uuid`: The UUID of the service to look for.

Returns: `true` if the service is present on the remote device, `false` if not. Note that for accurate results, a discovery of services should be done. If the UUID was in the advertisement, this might be true even before a full discovery.

Example:

```
if (peripheral.connect()) {
    peripheral.discoverAttributes();
    if (peripheral.hasService("180D")) {
        Serial.println("Heart Rate Service found on peripheral.");
    }
}
```

BLEDevice.hasCharacteristic()

Description: Checks if the remote device (when connected) has a characteristic with the given UUID (across all its services). This is useful if you know a specific characteristic UUID to look for. It requires that characteristics information is available (via discovery).

Syntax:

```
bool BLEDevice::hasCharacteristic(const char* uuid);
```

Parameters: - `uuid`: The UUID of the characteristic to search for.

Returns: `true` if a characteristic with this UUID exists on the remote device, `false` otherwise.

Example:

```
if (peripheral.hasCharacteristic("2A37")) { // Heart Rate Measurement char UUID
    Serial.println("Peripheral has Heart Rate Measurement characteristic.");
}
```

This assumes you have either performed `discoverAttributes()` or knew from advertisement or context that the service was present.

BLEDevice.serviceCount()

Description: Returns the number of GATT services discovered on the remote device. Only valid after connecting and performing a service discovery (e.g., via `discoverAttributes()` or similar means).

Syntax:

```
int BLEDevice::serviceCount();
```

Parameters: None.

Returns: Number of services on the remote device that have been discovered.

Example:

```
peripheral.discoverAttributes();
Serial.print("Services discovered: ");
Serial.println(peripheral.serviceCount());
```

BLEDevice.service()

Description: Retrieves a `BLEService` object representing one of the remote device's GATT services, by index or by UUID. This allows you to then explore its characteristics.

Syntax:

```
BLEService BLEDevice::service(int index);
BLEService BLEDevice::service(const char* uuid);
```

Parameters: - `index`: Index of the service (0 to `serviceCount()-1`). - `uuid`: UUID string of the desired service.

Returns: A `BLEService` object corresponding to the service at the index or matching the UUID. If the index is out of range or the UUID is not found, an invalid BLEService (evaluating to false) is returned.

Example (by index):

```
for (int i = 0; i < peripheral.serviceCount(); i++) {
  BLEService svc = peripheral.service(i);
  Serial.print("Service ");
  Serial.println(svc.uuid());
}
```

Example (by UUID):

```
BLEService batteryService = peripheral.service("180F");
if (batteryService) {
  Serial.println("Found Battery Service on remote device.");
}
```

BLEDevice.characteristicCount()

Description: Returns the total number of characteristics discovered on the remote device *across all services*. This count is available after discovering attributes.

Syntax:

```
int BLEDevice::characteristicCount();
```

Parameters: None.

Returns: The count of characteristics on the remote device.

Note: Characteristics are typically associated with specific services. It might be more common to iterate through services and then characteristics of each service via `BLEService.characteristicCount()`. This function provides a flat count of all characteristics.

BLEDevice.characteristic()

Description: Retrieves a `BLECharacteristic` from the remote device either by index (in a flat list of all characteristics) or by UUID (searching all services for that characteristic UUID).

Syntax:

```
BLECharacteristic BLEDevice::characteristic(int index);
BLECharacteristic BLEDevice::characteristic(const char* uuid);
```

Parameters: - `index`: Index of the characteristic (0 to `characteristicCount()-1`). - `uuid`: UUID of the desired characteristic.

Returns: A `BLECharacteristic` object corresponding to the characteristic. If the index is out of range or the UUID is not found, an invalid BLECharacteristic (evaluating to false) is returned.

Example (by UUID):

```
BLECharacteristic hrChar = peripheral.characteristic("2A37");
if (hrChar) {
  Serial.println("Heart Rate Measurement characteristic found!");
}
```

Usually you would know the service and characteristic; you could also navigate via `BLEService` objects (see BLEService.characteristic()) for clarity.

BLEDevice.connect()

Description: Initiates a connection to a remote peripheral device. This is used when the Arduino is acting as a central and wants to connect to a device discovered via scanning.

Syntax:

```
bool BLEDevice::connect();
```

Parameters: None.

Returns: `true` if the connection was successfully established, `false` if the connection attempt failed.

Usage: Only call on a BLEDevice that was obtained via scanning (`BLE.available()`) and is not yet connected. After a successful connection, you may want to call `discoverAttributes()` to enumerate services/characteristics.

Example:

```
BLEDevice peripheral = BLE.available();
if (peripheral) {
  Serial.println("Connecting...");
  if (peripheral.connect()) {
    Serial.println("Connected to peripheral!");
    peripheral.discoverAttributes();
    // Now can use peripheral.service() or characteristic()...
  } else {
    Serial.println("Connection failed.");
  }
}
```

BLEDevice.disconnect()

Description: Disconnects from the remote device. If the Arduino is central and connected to a peripheral, this breaks the connection. If the Arduino is peripheral, calling this on the BLEDevice representing the central will also disconnect that central.

Syntax:

```
bool BLEDevice::disconnect();
```

Parameters: None.

Returns: `true` if a disconnection command was issued (i.e., a connection was present), `false` if there was no connection to disconnect.

Example:

```
if (peripheral.connected()) {
  Serial.println("Disconnecting...");
  peripheral.disconnect();
}
```

After this, `peripheral.connected()` would return false (and on the other side, the device will see a disconnect).

BLEDevice.discoverAttributes()

Description: Performs a full discovery of the connected device's GATT services, characteristics, and descriptors. This populates the BLEDevice's internal lists of services and characteristics (and their descriptors). This is typically called after connecting to a peripheral, before attempting to use `hasService`, `characteristic()`, etc., unless you already know the structure from advertised data.

Syntax:

```
bool BLEDevice::discoverAttributes();
```

Parameters: None.

Returns: `true` if the discovery completed successfully, `false` if it failed (or no connection).

Behavior: This may take some time to run (depending on the number of services/characteristics on the remote device). It essentially queries all primary services, then for each service queries characteristics, and for each characteristic queries descriptors.

Example:

```
if (peripheral.connect()) {
  if (peripheral.discoverAttributes()) {
    Serial.println("Services and characteristics discovered.");
  }
}
```

After this, you can iterate over `peripheral.serviceCount()` or use `peripheral.characteristic("uuid")` to access what you need.

BLEDevice.poll()

Description: Polls the BLE subsystem for events related to this specific device and processes them. This is used to handle incoming data or events (like notifications, indications, or disconnection) for the connection. In practice, you should call `device.poll()` frequently in your loop for each active connection, especially when expecting notifications from a peripheral or to drive the internal event system.

Syntax:

```
void BLEDevice::poll();
void BLEDevice::poll(int timeout);
```

Parameters: - *(optional)* `timeout`: The maximum time in milliseconds to wait for an event. If omitted or 0, the poll will return immediately after checking events (non-blocking). If a positive timeout is provided, `poll()` will block up to that many milliseconds waiting for an event.

Returns: Nothing.

Usage: For a peripheral device connected to our Arduino, calling `centralDevice.poll()` will process any read/write requests or subscription events from that central. For our Arduino as a central connected to a peripheral, calling `peripheralDevice.poll()` will process incoming notifications or other events from that peripheral.

Example (Peripheral role):

```
BLEDevice central = BLE.central();
if (central) {
  // While central is connected, poll for events:
  central.poll();
  // e.g., this will trigger callbacks if the central writes to a characteristic
}
```

Example (Central role):

```
if (peripheral.connected()) {
  peripheral.poll();
  // e.g., this will process any notifications from the peripheral
  if (someChar.valueUpdated()) {
    // handle updated value...
  }
}
```

In many cases, calling `BLE.poll()` (which will poll all connections) or `central.poll()` in peripheral mode suffices. The library's documentation suggests using `BLEDevice.poll()` on the specific device especially when multiple connections are possible.

BLEDevice.connected()

Description: Indicates whether this BLEDevice is currently connected. For a BLEDevice that was obtained via scanning, this becomes true after a successful `connect()`. For a BLEDevice that represents a central (from `BLE.central()`), it will remain true while the central is connected.

Syntax:

```
bool BLEDevice::connected();
```

Parameters: None.

Returns: `true` if the BLEDevice is in a connected state, `false` if not connected.

Example:

```
if (peripheral.connected()) {
  // We have an active connection to the peripheral
} else {
  // Not connected (or disconnected)
}
```

This is similar to `BLE.connected()` but for a specific BLEDevice instance.

BLEService Class

The **BLEService** class represents a BLE GATT service. Services are collections of characteristics that perform related functions (for example, a Heart Rate Service contains heart rate measurement characteristics, etc.). In ArduinoBLE, BLEService is used both for **local services** (when the Arduino is acting as a peripheral offering services) and for **remote services** (when the Arduino is central and has discovered services on a peripheral).

A BLEService is primarily identified by a UUID. You typically create a BLEService when setting up your peripheral, add characteristics to it, and then add the service to the BLE stack via `BLE.addService()`. When acting as a central, you obtain BLEService objects by discovering them (e.g., via `BLEDevice.service(index)` or `BLEDevice.service(uuid)` after discovery).

Operator bool: A BLEService object evaluates to true if it represents a valid service (either a created local service or a discovered remote service). If it is uninitialized or invalid, it will evaluate to false.

BLEService() Constructor

Description: Creates a new BLEService object. There are multiple constructors available: - A default constructor for an empty/uninitialized BLEService. - A constructor that takes a UUID string to create a service with that UUID.

Syntax:

```
BLEService::BLEService();
BLEService::BLEService(const char* uuid);
```

Parameters (for UUID constructor): - `uuid`: A service UUID string. This can be a 16-bit UUID (e.g., `"180D"`) or a 128-bit UUID string (e.g., `"19B10000-E8F2-537E-4F6C-D104768A1214"`).

Returns: A BLEService instance.

Usage: For a local peripheral, you typically create a BLEService with the desired UUID and then add characteristics to it. For a remote service (central role), you usually obtain BLEService from a BLEDevice rather than constructing one directly.

Example (Peripheral):

```
BLEService ledService("19B10000-E8F2-537E-4F6C-
D104768A1214"); // Create service with a custom UUID
```

BLEService.uuid()

Description: Retrieves the UUID of the service.

Syntax:

```
String BLEService::uuid();
```

Parameters: None.

Returns: A `String` containing the service's UUID in standard format. For 16-bit UUIDs, it returns a 4-character string (e.g., `"180D"`), and for 128-bit it returns the full 36-character string with hyphens.

Example:

```
Serial.print("Service UUID: ");
Serial.println(myService.uuid());
```

BLEService.addCharacteristic()

Description: Adds a characteristic to this service (for a local service on a peripheral device). This links a `BLECharacteristic` to the service so that when the service is added to BLE (via `BLE.addService()`), the characteristics are included. This function is only used on the **peripheral side** (local service definition).

Syntax:

```
void BLEService::addCharacteristic(BLECharacteristic& characteristic);
```

Parameters: - `characteristic`: The `BLECharacteristic` object to add to the service.

Returns: Nothing.

Usage: Call this for each characteristic you want to belong to the service, before calling `BLE.addService()`. If the characteristic was previously associated with another service or already added, it may not be added again (and behavior is undefined). Typically, you create a characteristic with a UUID and properties, then add it.

Example:

```
BLEService ledService("19B10000-E8F2-537E-4F6C-D104768A1214");
BLECharacteristic switchChar("19B10001-E8F2-537E-4F6C-D104768A1214", BLERead |
BLEWrite);
ledService.addCharacteristic(switchChar);
BLE.addService(ledService);
```

This defines a service and adds a read/write characteristic to it.

BLEService.characteristic()

Description: For a discovered remote service (on a central), this function can retrieve a characteristic by UUID or by index within that service. This allows a central to get a `BLECharacteristic` object representing a remote characteristic of this service.

Syntax:

```
BLECharacteristic BLEService::characteristic(const char* uuid);
BLECharacteristic BLEService::characteristic(int index);
```

Parameters: - `uuid`: The UUID of the characteristic to retrieve. - `index`: The index of the characteristic within this service (0-based). The order is determined by discovery order.

Returns: A `BLECharacteristic` object for the requested characteristic, or an invalid BLECharacteristic if not found (or if called on a local service, which doesn't support this query).

Usage: Only applicable for BLEService objects that represent remote services (obtained from a BLEDevice after discovery). For local services, you already have direct references to characteristics you added.

Example:

```
BLEService remoteService = peripheral.service("180F"); // Battery Service
if (remoteService) {
   BLECharacteristic batteryLevelChar = remoteService.characteristic("2A19");
   if (batteryLevelChar) {
      batteryLevelChar.readValue(value);
      Serial.print("Battery level: ");
```

```
      Serial.print(value);
      Serial.println("%");
   }
}
```

In this example, after discovering attributes, we get the Battery Service by UUID and then get its Battery Level characteristic by UUID (`0x2A19`), then read its value.

BLEService.characteristicCount()

Description: Returns the number of characteristics that belong to this service. This is relevant for remote services discovered on a central.

Syntax:

```
int BLEService::characteristicCount();
```

Parameters: None.

Returns: The count of characteristics within the service.

Usage: Only valid for a BLEService representing a remote service after discovery. For local services, you typically know how many you added and manage them in code without querying.

BLEService – Operator bool

As noted, `BLEService` can be evaluated in a boolean context. It will be `true` (or `if (service)` will pass) if the service object is valid (either a defined local service or a discovered remote service). If a service object was default-constructed or a lookup (by UUID or index) failed, it will evaluate as false.

BLECharacteristic Class

The **BLECharacteristic** class represents a BLE GATT characteristic, which holds a value and properties (such as Read, Write, Notify, Indicate). In ArduinoBLE, `BLECharacteristic` is used for both: - **Local characteristics**: when the Arduino is a peripheral, you create BLECharacteristic objects, add them to services, and set their values or respond to reads/writes. - **Remote characteristics**: when the Arduino is a central, you obtain BLECharacteristic objects (via `BLEDevice.characteristic()` or `BLEService.characteristic()`) for characteristics on the peer, and you can read or write their values, or subscribe to notifications.

A BLECharacteristic has a UUID, a set of properties (capabilities), and potentially one or more descriptors (like the Client Characteristic Configuration descriptor, which is automatically managed if Notify/Indicate is used, and optional User Description descriptors etc.).

Note on creation: To create a local characteristic, you typically specify a UUID, properties, and a value length or initial value. ArduinoBLE provides some convenience subclasses for common data types (e.g., `BLEByteCharacteristic`, `BLEIntCharacteristic`, etc.), but here we describe using the generic BLECharacteristic.

Properties: Properties define how the characteristic can be used. Use bitwise OR of the provided property flags: - `BLEBroadcast` – permit broadcasting (not commonly used). - `BLERead` – permit reads of the value. - `BLEWrite` – permit writes to the value (without response). - `BLEWriteWithoutResponse` – permit writes without requiring a response (often combined with BLEWrite). - `BLENotify` – permit notifications of value changes. - `BLEIndicate` – permit indications of value changes (like notify but with acknowledgment).

For remote characteristics, properties indicate what operations are supported by that characteristic on the peripheral.

Event Handlers: For local characteristics, you can set event handlers (callbacks) to be notified when a central performs certain actions (writes, subscribes, unsubscribes, etc.) on the characteristic.

Operator bool: A BLECharacteristic object evaluates to true if it is valid (either a created local characteristic or a discovered remote characteristic). If a characteristic lookup fails or an object is default-constructed, it will evaluate to false.

BLECharacteristic() Constructors

Description: Constructs a BLECharacteristic object. Several constructor overloads exist for creating local characteristics: - With specified UUID, properties, and value size. - Possibly with an initial value provided (for convenience if the value is known at startup). - There are also convenience constructors for certain types (e.g., providing a `const char*` initial value or using the specialized classes for bytes, etc.).

Syntax:

```
BLECharacteristic(const char* uuid, uint8_t properties, int valueSize, bool
fixedLength = false);
BLECharacteristic(const char* uuid, uint8_t properties, const uint8_t value[],
int valueSize, bool fixedLength = false);
BLECharacteristic(const char* uuid, uint8_t properties, const char*
initialValue);
```

Parameters: - `uuid`: The UUID for the characteristic. - `properties`: A combination of property flags (e.g., `BLERead | BLENotify`). - `valueSize`: The maximum size in bytes of the characteristic's value. - `fixedLength`: If true, the characteristic's value length is fixed at `valueSize`. If false, the value can be variable length up to `valueSize`. - `value[]`: An initial value byte array. - `initialValue`: An initial value provided as a C-string (convenience for fixed-length text or descriptors).

Returns: A BLECharacteristic object ready to be added to a service (for local usage).

Usage: For local characteristics, after constructing, you typically add it to a BLEService with `service.addCharacteristic(char)`. For remote characteristics, you do not construct them; instead, you get them from a BLEDevice or BLEService as shown earlier.

Example (Local char with no initial value):

```
BLECharacteristic tempChar("2A6E",  // Temperature Measurement UUID
                           BLERead | BLENotify,
                           2);      // 2 bytes (e.g., temperature in 0.01°C
units)
```

Example (Local char with initial value):

```
const char* initialValue = "Hello";
BLECharacteristic greetChar("19B10001-E8F2-537E-4F6C-D104768A1214",
                            BLERead | BLEWrite,
                            initialValue);
```

BLECharacteristic.uuid()

Description: Gets the UUID of the characteristic.

Syntax:

```
String BLECharacteristic::uuid();
```

Parameters: None.

Returns: The UUID string of the characteristic.

Example:

```
Serial.println(characteristic.uuid());
```

BLECharacteristic.properties()

Description: Returns the properties bitmask of the characteristic, indicating its capabilities (readable, writable, etc.).

Syntax:

```
uint8_t BLECharacteristic::properties();
```

Parameters: None.

Returns: An 8-bit bitmask of property flags. Use the provided constants to interpret (e.g., mask with `BLERead`, `BLEWrite`, `BLENotify`, etc., to check individual properties).

Example:

```
uint8_t props = characteristic.properties();
if (props & BLERead) {
  Serial.println("Characteristic is readable.");
}
```

BLECharacteristic.value()

Description: Retrieves a pointer to the current value bytes of the characteristic. For a local characteristic, this is the current value stored on the peripheral. For a remote characteristic (on central), this would be the last known value (perhaps after a read or notification).

Syntax:

```
const uint8_t* BLECharacteristic::value();
```

Parameters: None.

Returns: A pointer to a buffer containing the characteristic's value bytes. The length of the value can be obtained with `valueLength()`. If no value is present, it may return `nullptr` or an empty buffer.

Usage: This is often used on the peripheral side inside event handlers (e.g., after a central writes new data, you could inspect the updated value via this pointer). On the central side, after calling `readValue()` or when a notification arrives (setting `valueUpdated()`), you could use this to get the data.

Example (Peripheral side, reading a written value):

```
void onWritten(BLEDevice central, BLECharacteristic characteristic) {
  // This callback is called when the central writes to this characteristic
  const uint8_t* data = characteristic.value();
  int length = characteristic.valueLength();
  Serial.print("New value written (length ");
  Serial.print(length);
```

```
  Serial.println("):");
  for (int i = 0; i < length; i++) {
    Serial.print(data[i], HEX);
    Serial.print(" ");
  }
  Serial.println();
}
```

(You would set this function as an event handler for BLEWritten, see setEventHandler.)

BLECharacteristic.valueLength()

Description: Gets the length (in bytes) of the current value of the characteristic.

Syntax:

```
int BLECharacteristic::valueLength();
```

Parameters: None.

Returns: The number of bytes currently in the characteristic's value.

Example: (See the example in `value()` above, where `characteristic.valueLength()` is used.)

BLECharacteristic.valueSize()

Description: Gets the maximum size (in bytes) that the characteristic's value can hold.

Syntax:

```
int BLECharacteristic::valueSize();
```

Parameters: None.

Returns: The maximum capacity of the value (as set when the characteristic was created, or a default size if not explicitly set for remote characteristics).

Example:

```
Serial.print("Max value size: ");
Serial.println(characteristic.valueSize());
```

For a local characteristic, this is the size you specified in the constructor. For a remote characteristic, this might be the maximum size as declared by the peripheral, if known.

BLECharacteristic.readValue()

Description: Reads the current value of the characteristic. For a **remote characteristic** (when acting as central), this will send a read request to the peripheral if not already done, or return a cached value if available. For a **local characteristic** (on peripheral), since the value is stored locally, this essentially copies the current value into the provided buffer or variable.

Syntax:

```
int BLECharacteristic::readValue(uint8_t buffer[], int length);
int BLECharacteristic::readValue(<T> &value);
```

(Here `<T>` *can be a primitive type or array; the function template allows reading into a variable of that type by reference.)*

Parameters: - `buffer`: Byte array to copy the value into. - `length`: Size of the buffer (number of bytes to read at most). - `value`: A variable passed by reference to receive the data (for example, a `byte`, `int`, or other type that matches the characteristic's data size).

Returns: The number of bytes read and copied into the buffer/variable. This will be the length of the characteristic's value or the buffer length, whichever is smaller. If the read fails (for remote device), it may return 0.

Example (Central, reading remote value into a buffer):

```
uint8_t data[8];
int bytesRead = remoteChar.readValue(data, sizeof(data));
Serial.print("Read ");
Serial.print(bytesRead);
Serial.println(" bytes from characteristic.");
```

Example (Central, reading into a simple variable):

```
byte level;
if (batteryLevelChar.readValue(level)) {
  Serial.print("Battery level is ");
  Serial.print(level);
  Serial.println("%");
}
```

In the above, if `batteryLevelChar` is a one-byte value characteristic, we can directly read it into a `byte`.

Note: On a peripheral, since you already have local access to the value, you might not often call `readValue()` – you could use `value()` and `valueLength()` directly. `readValue()` is most useful in central mode or for a uniform way to get the value.

BLECharacteristic.writeValue()

Description: Writes a new value to the characteristic. If this is a **local characteristic** on a peripheral, it updates the local value and, if there are subscribed centrals, will send notifications/indications as appropriate. If this is a **remote characteristic** on a central, this will send a write request or command to the peripheral to update the value on that device.

Syntax:

```
int BLECharacteristic::writeValue(const uint8_t value[], int length);
int BLECharacteristic::writeValue(const T& value);
```

(There are overloads allowing you to pass a byte array with specified length, or a single value/variable of type T (e.g., byte, int, etc., or even a string).)

Parameters: - `value[]`: Byte array containing the data to write. - `length`: Number of bytes from the array to write. - `value`: A value or variable to write (the library provides overloads for standard data types and also Arduino `String` or `const char*`).

Returns: `1` on success, `0` on failure. (Note: This return is a boolean-like success indicator, *not* the number of bytes written.)

Behavior: If writing to a remote characteristic, the function will use a Write Request (with response) by default. Some characteristics might permit write without response; in such a case, the library will use a write command if possible (especially if `withResponse` is set to false in an internal overload). However, by default using this function tries to ensure the write succeeds (so counts as a success if acknowledged by the peripheral).

Example (Peripheral, updating a local char and notifying central):

```
// Suppose temperatureChar is a BLENotify characteristic
int temperature = readTemperatureSensor();  // some function returns int
uint8_t tempData[2];
tempData[0] = temperature & 0xFF;
tempData[1] = (temperature >> 8) & 0xFF;
temperatureChar.writeValue(tempData, 2);  // Update characteristic value
// If a central is subscribed, they will get a notification with new value
```

Example (Central, writing to a remote char):

```
if (ledSwitchChar.writeValue((byte)0x01)) {
  Serial.println("Sent command to turn LED on.");
} else {
  Serial.println("Failed to write to LED switch characteristic.");
}
```

In this example, `ledSwitchChar` is a remote characteristic that controls an LED on the peripheral. We write a single byte `0x01` to turn it on. The return value indicates if the write was successful.

BLECharacteristic.setEventHandler()

Description: Sets a callback function (event handler) that will be called when a specified event occurs on this characteristic. This is used only for **local characteristics** on a peripheral. Typical events include: - A central *writing* to the characteristic (event type `BLEWritten`). - A central *reading* the characteristic (event type `BLERead`, if you want to take action when a read request comes in). - A central *subscribing* to notifications/indications (event type `BLESubscribed`). - A central *unsubscribing* (event type `BLEUnsubscribed`).

Using event handlers allows your sketch to react in real-time to BLE interactions.

Syntax:

```
void BLECharacteristic::setEventHandler(BLECharacteristicEvent event,
BLECharacteristicEventHandler callback);
```

Parameters: - `event`: The event type to handle. This is an enum or constant that can be `BLEWritten`, `BLERead`, `BLESubscribed`, `BLEUnsubscribed`. - `callback`: A function to call when the specified event occurs. The function should match the signature: `void callback(BLEDevice central, BLECharacteristic characteristic)` for write/subscribe events, or `void callback(BLEDevice central, BLECharacteristic characteristic, uint8_t offset)` for read events (the library might handle read differently, often you might not need a handler for read unless you generate value on the fly).

Returns: Nothing.

Example:

```
void onLEDWritten(BLEDevice central, BLECharacteristic chr) {
  // central wrote new value to the LED switch characteristic
  byte val;
  chr.readValue(val);
```

```
  Serial.print("Central wrote: ");
  Serial.println(val);
  digitalWrite(LED_PIN, val ? HIGH : LOW);
}

...

// Setting up the characteristic:
BLECharacteristic ledSwitchChar("19B10001-E8F2-537E-4F6C-D104768A1214", BLERead
| BLEWrite);
ledSwitchChar.setEventHandler(BLEWritten, onLEDWritten);
```

In this snippet, whenever a connected central writes to `ledSwitchChar`, the function `onLEDWritten` will be invoked. Inside it, we read the new value and update a local LED.

BLECharacteristic.subscribe()

Description: Subscribes to notifications (and/or indications) from a remote characteristic. This is used on the **central side**. It effectively writes the Client Characteristic Configuration Descriptor (CCCD) on the remote device to enable notifications/indications for this characteristic.

Syntax:

```
bool BLECharacteristic::subscribe();
```

Parameters: None.

Returns: `true` if the subscription (enabling notifications/indications) was successful, `false` otherwise.

Conditions: The characteristic must have the Notify and/or Indicate property (you can check with `canSubscribe()` before calling). Also, you should be connected to the device and have discovered descriptors.

Example:

```
if (heartRateChar.canSubscribe()) {
  if (heartRateChar.subscribe()) {
    Serial.println("Subscribed to heart rate notifications.");
  }
}
```

After subscribing, whenever the peripheral updates that characteristic and sends a notification/indication, the ArduinoBLE library will update the `heartRateChar` value internally and mark it as updated (so

`valueUpdated()` becomes true), and/or call an event handler if one is set for value update (on central typically you use polling or valueUpdated).

BLECharacteristic.unsubscribe()

Description: Unsubscribes from notifications/indications on a remote characteristic. This will disable further notifications from the peripheral for this characteristic by writing to the CCCD.

Syntax:

```
bool BLECharacteristic::unsubscribe();
```

Parameters: None.

Returns: `true` if successfully unsubscribed (or if not currently subscribed), `false` if an error occurred.

Example:

```
heartRateChar.unsubscribe();
Serial.println("Unsubscribed from heart rate notifications.");
```

BLECharacteristic.subscribed()

Description: Checks if the local side is subscribed to notifications for this characteristic. For a **central**, this returns true if `subscribe()` was called and notifications are active. For a **peripheral's local characteristic**, this returns true if at least one connected central has subscribed to this characteristic's notifications/indications.

Syntax:

```
bool BLECharacteristic::subscribed();
```

Parameters: None.

Returns: `true` if subscribed, `false` if not.

Example (Peripheral):

```
if (batteryLevelChar.subscribed()) {
  // At least one central is listening for battery level notifications
  // We might send periodic updates:
```

```
  batteryLevelChar.writeValue(currentLevel);
}
```

Example (Central):

```
if (!remoteChar.subscribed()) {
  remoteChar.subscribe();
}
```

BLECharacteristic.canRead()

Description: Indicates if the characteristic has the Read property. For a local characteristic, this means you allowed BLERead when creating it. For a remote characteristic, this reflects the peer's capabilities.

Syntax:

```
bool BLECharacteristic::canRead();
```

Parameters: None.

Returns: `true` if the characteristic is readable, `false` otherwise.

Example:

```
if (remoteChar.canRead()) {
  remoteChar.readValue(buffer, length);
}
```

BLECharacteristic.canWrite()

Description: Indicates if the characteristic supports Write (with response) by a client/central.

Syntax:

```
bool BLECharacteristic::canWrite();
```

Returns: `true` if the characteristic has write permission (write with response), `false` otherwise.

Example:

```
if (remoteChar.canWrite()) {
  remoteChar.writeValue(data, len);
}
```

For a local characteristic on a peripheral, `canWrite()` being true means centrals are allowed to write to it.

BLECharacteristic.canSubscribe()

Description: Indicates if the characteristic supports subscription (i.e., has Notify or Indicate property that a central can enable).

Syntax:

```
bool BLECharacteristic::canSubscribe();
```

Returns: `true` if the characteristic has Notify or Indicate capability, `false` otherwise.

Example:

```
if (remoteChar.canSubscribe()) {
  remoteChar.subscribe();
}
```

BLECharacteristic.canUnsubscribe()

Description: Indicates if unsubscribing is applicable. In practice this will usually return the same as `canSubscribe()` (if you can subscribe, you can also unsubscribe). It might specifically check if currently subscribed and thus can unsubscribe.

Syntax:

```
bool BLECharacteristic::canUnsubscribe();
```

Returns: `true` if the characteristic is currently in a state that can be unsubscribed (e.g., was subscribed and supports it), otherwise `false`.

Example:

```
if (remoteChar.subscribed() && remoteChar.canUnsubscribe()) {
  remoteChar.unsubscribe();
}
```

BLECharacteristic.hasDescriptor()

Description: Checks if this characteristic has a descriptor with a given UUID. This is typically used on a **remote characteristic** after discovery, to see if certain descriptors (like a User Description `0x2901`, or others) are present. (On a local characteristic, you know what descriptors you added manually, aside from the CCCD which is auto-managed but not exposed the same way.)

Syntax:

```
bool BLECharacteristic::hasDescriptor(const char* uuid);
```

Parameters: - `uuid`: UUID of the descriptor to find (e.g., `"2901"` for Characteristic User Description).

Returns: `true` if the descriptor is present on this characteristic, `false` if not.

Example:

```
if (remoteChar.hasDescriptor("2901")) {
  BLEDescriptor desc = remoteChar.descriptor("2901");
  desc.read(); // Read the descriptor (likely user description)
  Serial.println((char*)desc.value()); // print descriptor as string
}
```

BLECharacteristic.descriptorCount()

Description: Returns the number of descriptors that this characteristic has. Only relevant for a remote characteristic after descriptors have been discovered.

Syntax:

```
int BLECharacteristic::descriptorCount();
```

Returns: Number of descriptors for this characteristic.

Example:

```
Serial.print("Descriptors for char UUID ");
Serial.print(char.uuid());
Serial.print(": ");
Serial.println(char.descriptorCount());
```

BLECharacteristic.descriptor()

Description: Retrieves a `BLEDescriptor` associated with this characteristic, either by index or by UUID. For local characteristics, you typically use references to descriptors you added. For remote, you use this after discovery.

Syntax:

```
BLEDescriptor BLECharacteristic::descriptor(int index);
BLEDescriptor BLECharacteristic::descriptor(const char* uuid);
```

Parameters: - `index`: Index of descriptor (0 to `descriptorCount()-1`). - `uuid`: UUID of the descriptor.

Returns: A `BLEDescriptor` object if found, otherwise an invalid BLEDescriptor.

Example (by UUID):
See the example under `hasDescriptor()` above.

Example (by index):

```
for (int j = 0; j < char.descriptorCount(); j++) {
   BLEDescriptor desc = char.descriptor(j);
   Serial.print("Descriptor UUID: ");
   Serial.println(desc.uuid());
}
```

BLECharacteristic.written()

Description: For a local characteristic on a peripheral, this indicates if the characteristic value has been written by a central since the last time this flag was checked. This can be used in a loop to poll for new writes if you prefer not to use callbacks.

Syntax:

```
bool BLECharacteristic::written();
```

Returns: `true` if the characteristic was written (and not yet handled) by a central, `false` otherwise. Reading this might clear the flag until the next write (the documentation suggests it's an event-style flag).

Example:

```
// In loop, peripheral side:
if (ledSwitchChar.written()) {
  byte newValue;
  ledSwitchChar.readValue(newValue);
  digitalWrite(LED_PIN, newValue ? HIGH : LOW);
  Serial.println("LED state updated via central write.");
}
```

In this example, instead of using an event handler, the code polls `written()` to detect if the central wrote to `ledSwitchChar`, then reads the new value and acts on it.

BLECharacteristic.valueUpdated()

Description: For a remote characteristic (on a central), this indicates if the characteristic's value has been updated due to a notification or indication from the peripheral since the last check. This allows a central to poll for new data without continuously reading.

Syntax:

```
bool BLECharacteristic::valueUpdated();
```

Returns: `true` if a new value was received (via notification/indication) since the last time `valueUpdated()` was checked (it typically resets to false once you call this or read the value), `false` otherwise.

Example:

```
if (heartRateChar.valueUpdated()) {
  uint8_t hr;
  heartRateChar.readValue(hr);
  Serial.print("Heart Rate update: ");
  Serial.println(hr);
}
```

This would be in a loop for a central that has subscribed to heart rate notifications. Each time a new heart rate measurement is notified, `valueUpdated()` becomes true so you can read the latest value.

BLECharacteristic.addDescriptor()

Description: Adds a descriptor to a local characteristic (peripheral side). This allows you to attach an additional descriptor, such as a **User Description (UUID 0x2901)** or a **Presentation Format (UUID 0x2904)**, to the characteristic. The CCCD (Client Characteristic Configuration Descriptor, 0x2902) for Notify/Indicate is automatically managed by the library when you use those properties, so you do not add that manually.

Syntax:

```
void BLECharacteristic::addDescriptor(BLEDescriptor& descriptor);
```

Parameters: - `descriptor`: The BLEDescriptor object to add.

Returns: Nothing.

Usage: Must be done before starting BLE (or at least before starting advertising) and after the descriptor and characteristic are created. Add all desired descriptors to the characteristic so that when the service is added to BLE, these descriptors are included.

Example:

```
BLEDescriptor userDesc("2901", "LED Switch"); // 0x2901 is User Description
descriptor
ledSwitchChar.addDescriptor(userDesc);
```

Here we add a user-readable description "LED Switch" to the `ledSwitchChar` characteristic. A central can read this descriptor to get a textual description of what the characteristic does.

BLECharacteristic – Operator bool

Like other BLE classes, a BLECharacteristic object can be used in a boolean context. It evaluates to `true` if it references a valid characteristic (one that has been created or discovered), or `false` if it is not valid (for example, the result of a failed lookup or a default-constructed object).

BLEDescriptor Class

The **BLEDescriptor** class represents a BLE GATT descriptor. Descriptors provide additional metadata or configuration for a characteristic. Common descriptors include: - Characteristic User Description (UUID 0x2901): a textual description of the characteristic's purpose. - Client Characteristic Configuration (UUID

0x2902): used to enable/disable notifications or indications (this one is typically automatically handled by the BLE stack; you usually don't manually manipulate the CCCD via this class in ArduinoBLE – instead you use `subscribe()` / `unsubscribe()` on the characteristic). - Other BLE specification descriptors like Characteristic Presentation Format (0x2904), etc., or custom descriptors with custom UUIDs.

In ArduinoBLE, you use BLEDescriptor mainly to: - Create and add a descriptor to a local characteristic (e.g., a user description). - Access and read a descriptor on a remote characteristic (for example, reading the user description or presentation format from a peripheral).

BLEDescriptor() Constructors

Description: Constructs a BLEDescriptor. There are a couple of constructors provided: - One that takes a UUID and a byte array value (with length). - One that takes a UUID and a C-string as the value (convenient for descriptors that are textual).

Syntax:

```
BLEDescriptor(const char* uuid, const uint8_t value[], int valueSize);
BLEDescriptor(const char* uuid, const char* value);
```

Parameters: - `uuid`: The UUID of the descriptor (e.g., `"2901"` for user description, or a custom 128-bit UUID). - `value[]`: An initial value byte array for the descriptor. - `valueSize`: Length of the byte array. - `value` (string): An initial value provided as a C-string (the library will take the string's bytes as the descriptor value). This is useful for user description, since it's a human-readable text.

Returns: A BLEDescriptor object that can be added to a characteristic (if local) or used to hold a remote descriptor.

Usage: - For local descriptors: create it and then call `characteristic.addDescriptor(descriptor)`. - For remote descriptors: you typically obtain BLEDescriptor objects from `BLECharacteristic.descriptor()` rather than constructing them.

Example (local User Description descriptor):

```
BLEDescriptor userDescDescriptor("2901", "Temperature in Celsius");
temperatureChar.addDescriptor(userDescDescriptor);
```

BLEDescriptor.uuid()

Description: Gets the UUID of the descriptor.

Syntax:

```
String BLEDescriptor::uuid();
```

Parameters: None.

Returns: The UUID of the descriptor as a string.

Example:

```
Serial.println(descriptor.uuid());
```

BLEDescriptor.value()

Description: Provides direct access to the value of the descriptor as a byte array pointer.

Syntax:

```
const uint8_t* BLEDescriptor::value();
```

Parameters: None.

Returns: Pointer to the descriptor's value data (const, since you typically modify via write functions). For a remote descriptor, this would be populated after reading the descriptor. For a local descriptor, this is the value you set (e.g., via the constructor or if there were a way to update it).

Example: After reading a descriptor on a remote device:

```
descriptor.read();
int len = descriptor.valueLength();
const uint8_t* val = descriptor.value();
Serial.print("Descriptor raw value (len ");
Serial.print(len);
Serial.println("):");
for (int i=0; i<len; i++) {
  Serial.print(val[i], HEX);
  Serial.print(" ");
}
Serial.println();
```

If the descriptor was a user description (text), you could also interpret it as a string:

```
Serial.println((const char*)descriptor.value());
```

BLEDescriptor.valueLength()

Description: Returns the length (number of bytes) of the descriptor's value.

Syntax:

```
int BLEDescriptor::valueLength();
```

Parameters: None.

Returns: Length of the value in bytes.

Example: (See example in `value()` above.)

BLEDescriptor.valueSize()

Description: Returns the maximum size of the descriptor's value (for local descriptors, this is the size of the initial value provided; for remote, it could be the maximum the peer allows, though usually descriptors have small fixed sizes).

Syntax:

```
int BLEDescriptor::valueSize();
```

Parameters: None.

Returns: The capacity in bytes of the descriptor's value.

Example:

```
Serial.print("Descriptor max size: ");
Serial.println(descriptor.valueSize());
```

For a user description that was set via a string of 18 characters, `valueSize()` might be 18 (or 19 including a null terminator if that was counted – typically the library will store the exact bytes given).

BLEDescriptor.read()

Description: Initiates a read of the descriptor's value from a remote device. This is used on a **central** to read a descriptor of a peripheral's characteristic. For a local descriptor on a peripheral, reading is handled by the BLE stack when a central reads it, so this function is primarily for central usage.

Syntax:

```
bool BLEDescriptor::read();
```

Parameters: None.

Returns: `true` if the read request was successfully executed (and the value has been updated), `false` if it failed.

Effect: On success, the descriptor's internal value buffer is updated with the data from the peripheral, and `valueLength()` will reflect the number of bytes read.

Example:

```
BLEDescriptor userDesc = someChar.descriptor("2901");
if (userDesc.read()) {
    Serial.print("Description: ");
    Serial.println((char*)userDesc.value());
}
```

This attempts to read the user description descriptor of a remote characteristic and print it as a string.

BLEDescriptor.readValue()

Description: Reads the descriptor's value into a provided buffer or variable. This is a convenience that combines `read()` and accessing the value. It will perform the read (for remote descriptors) and copy the result.

Syntax:

```
int BLEDescriptor::readValue(uint8_t buffer[], int length);
int BLEDescriptor::readValue(T& value);
```

Parameters: - `buffer`: Byte array to fill with the descriptor's value. - `length`: Size of the buffer. - `value`: A variable to fill (for example, a `String` or other type if supported, or a byte for a 1-byte descriptor).

Returns: The number of bytes read into the buffer/variable. Returns 0 if read failed or no bytes available.

Example:

```
char descStr[20];
int n = userDesc.readValue((uint8_t*)descStr, sizeof(descStr));
if (n > 0) {
  descStr[n] = '\0'; // ensure null-terminated if within buffer
  Serial.print("User description: ");
  Serial.println(descStr);
}
```

For descriptors that are simple (like a 1-byte format descriptor), you might do:

```
uint8_t format;
if (formatDesc.readValue(format)) {
   Serial.print("Format value: 0x");
   Serial.println(format, HEX);
}
```

BLEDescriptor – Operator bool

A BLEDescriptor object will evaluate to `true` if it is valid (either a local descriptor created or a remote descriptor discovered). If a BLEDescriptor is default-constructed or a lookup (by uuid/index) failed, it will evaluate as `false`. Always check the descriptor in boolean context before using it to ensure it's valid.

Example:

```
BLEDescriptor desc = char.descriptor("2904");
if (desc) {
   desc.read();
   // use desc...
}
```

This concludes the ArduinoBLE library API documentation for the BLE, BLEDevice, BLEService, BLECharacteristic, and BLEDescriptor classes. With these classes, you can create BLE peripherals with custom services and characteristics, or build BLE central applications that scan for devices and interact with their GATT servers. Use this reference as a guide to implementing BLE functionality in your Arduino projects.
