Bluetooth BLE

Using the low level ESP-IDF BLE functions can be considered difficult. There are a lot of concepts, there is a lot of state flowing around and events needing handled. This is an ideal candidate for a class level encapsulation.

We'll split the story into two parts … being a BLE Server and being a BLE Client.

A BLE Server

A BLE Server is also known as a BLE Peripheral. It is likely that this is how you will most commonly use the ESP32. With a BLE Server, you will expose one or more services where each service has one or more characteristics and each characteristic may have zero or more descriptors. I think we'll agree that is quite a lot going on.

In our C++ model, we create the concept of classes that represent these items.

- `BLEServer` – Models a server.
- `BLEService` – Models a service.
- `BLECharacteristic` – Models a characteristic.
- `BLEDescriptor` – Models a descriptor.
- `BLEAdvertising` – Models advertising.

At a high level, the pseudo code becomes:

```
BLEServer* pServer   = new BLEServer();
BLEService* pService = pServer->createService(ServiceUUID);
BLECharacteristic* pCharacteristic =
   pService->createCharacteristic(CharacteristicUUID);
pCharacteristic->setValue("Hello world");
pService->start();
```

Hopefully you see how this fits together … we create the service, create a characteristic on the service, set a value for the characteristic and then ask the service to start responding to requests.

If you are familiar with the BLE APIs, you will likely notice what is missing … all the complexity and glue code necessary for event handling and processing of BLE requests. All of this is *handled for you* by the implementation of the classes. This means that you can focus on your intent/usage of BLE while keeping your *plumbing* to a minimum.

The goal of these classes is to efficiently process BLE workflow encapsulating the plumbing so you don't need to worry about it … but within is the danger that the implementation will restrict you from some tasks by hiding functions that you might otherwise have needed for your own project. Thankfully, this has not been seen to be the case. The classes expose simple high level APIs for the 95% of common practices while at the same time providing methods that can be called to tweak and tailor the operations for the rarer cases. Hopefully you won't need those often but, if and when you do, they are there for you.

When a BLE Server is running, what must happen next is that peer devices (clients) must be able to locate it. This is made possible through the notion of advertising. The BLE Server can broadcast its existence along with sufficient information to allow a client to know what services it can provide.

Once we have started the BLE Server, we can ask it for an object (`BLEAdvertising`) that owns the advertisements that the server produces:

```
BLEAdvertising* pAdvertising = pServer->getAdvertising();
pAdvertising->start();
```

Once performed, the server can be found by the clients.

A core concept of BLE is the notion of the characteristic. Think of this as a stateful *record* that has an identity and a value. A peer device (if permitted) can read the value of the characteristic or set a new value. This is the core notion of BLE. The value read from the characteristic provides information. For example, if the characteristic represents your heart-rate measured from a sensor, then a remote client can retrieve your current heart rate by reading the characteristic value. Alternatively, the characteristic maintained in the BLE Server may represent the state of something that can be activated or changed. As another example, imagine the characteristic represents the state of your car door's lock. You can read the characteristic to determine that your door is locked when you leave or, conversely, you can set the characteristics value to an unlocked state which would result in the server mechanically unlocking the door.

To model these notions in our C++ classes, we leverage the notion that a C++ class can be sub-classed to be more specialized. A class called `BLECharacteristicCallbacks` provides two methods that can be overriden:

- `onRead(BLECharacteristic* pCharacteristic)` – Called when a read request arrives. A new value of the characteristic can be set before return.
- `onWrite(BLECharacteristic* pCharacteristic)` – Called when a write request arrives. The new value has been set in the characteristic already.

To utilize, we can then override these functions:

```
class MyCallback: public BLECharateristicCallback {
   void onRead(BLECharacteristic *) {
      // Do something before the read completes.
   }

   void onWrite(BLECharacteristic *) {
      // Do something because a new value was written.
   }
}
```

To use this technique, we inform our `BLECharacteristic` about a callback handler. For example:

```
pCharacteristic->setCallbacks(new MyCallback());
```

Here is an example that sends the time since startup each time a peer requests the value:

```
class MyCallbackHandler: public BLECharacteristicCallbacks {
   void onRead(BLECharacteristic *pCharacteristic) {
      struct timeval tv;
      gettimeofday(&tv, nullptr);
      std::ostringstream os;
      os << "Time: " << tv.tv_sec;
      pCharacteristic->setValue(os.str());
   }
};
```

Similar to the characteristic callbacks, we also have server callbacks. These detect a connection and a disconnection. These are subclassed from the `BLEServerCallbacks` class which has virtual methods for:

- `onConnect(BLEServer* pServer)` – Called when a connection occurs.
- `onDisconnect(BLEServer* pServer)` – Called when a disconnection occurs.

Another consideration for our BLE Server is the idea that we might want to "push" data to the peer when something interesting happens. Up till now we have considered the idea that we can receive read requests to get the current value or can receive write requests to set the current value. There is one more operation that we are interested in that is invoked on the server side. That operation is called "indicate". It is used to signal (or indicate) to the peer that the characteristic value has changed. The peer will receive an indication event to let it know that the change has occurred.

On the server, we can cause an indication to occur by invoking:

```
pCharacteristic->indicate();
```

The current value of the characteristic will be the value transmitted to the peer. We might pair this call with a previous request to set a new value:

```
pCharacteristic->setValue("HighTemp");
pCharacteristic->indicate();
```

A similar function to `indicate()` is called `notify()`. The distinction between them is that an `indicate()` receives a confirmation while a `notify()` does not receive a confirmation.

There are other events that we might need to be cognizant off. These include the connection event detected when a peer forms a connect against us. This is associated with a server (`BLEServer` class) and results in the `onConnection()` method being called.

A BLE Client

Now we will turn our attention to being a client. In this story, our ESP32 doesn't host services but instead wishes to be a consumer of services hosted elsewhere. The story can further be broken down into two sub-stories … namely scanning and interaction.

If we assume that our ESP32 starts up and wishes to be a client of a remote BLE server then it has to connect to that server. In order to connect to the server, we need to know the address of the server. An address is a 6 byte value commonly written in the form

```
nn:nn:nn:nn:nn:nn
```

While in principle this can be hard-coded into your application or manually entered through some interaction story this is not the common practice. Instead, we perform a procedure known as a *scan*. Scanning is the idea that we actively listen on the BLE radio frequencies for servers which are actively advertising their existence. As an analogy, imagine we enter a dark room and we have no idea who is in there with us. We take off our ear-muffs (yes, for some reason we entered a dark room with no light at all wearing ear-muffs … but then, this is just an analogy) and we start hearing voices. We hear "Hi this is Bob, I can make toast" and "Hi this is Susan, I can tell you what temperature it is" and we also hear "Hi this is Brian, ask me what I can do". In this analogy, Bob, Susan and Brian are the addresses of the BLE devices. Each of them are continually saying out loud that they exist and, in some cases, what they can do for us.

When we perform a BLE scan, we receive short records of data that always contain the address of the advertiser and *sometimes* additional information such as the services they provide or other descriptive items. This information arrives at us passively. All we need do is listen and we learn. Should we need to learn more about the devices, we can connect to them (even the ones that told us little) and explicitly ask them (in more detail) what they can do.

And …. that is the principle of scanning. In our C++ BLE story, we have modeled this through the C++ class called `BLEScan`. We get an instance of this class by asking the BLE device for it using:

```
BLEScan* pMyScan = BLE::getScan();
```

The object returned to us is a singleton. This was chosen because we will only want to be scanning once per ESP32 at any given time.

The `BLEScan` object when presented to us isn't actually yet scanning, to start it scanning we invoke its start() method passing in how long we would like it to scan for. This is an interval measured in seconds. For example:

```
pMyScan.start(30); // Scan for 30 seconds
```

This is a non-blocking call. The scan starts immediately and will occur in the back ground. Since we are scanning, we will want to know about the devices that we find and this is where a callback function comes into play. Before calling start() we will want to register a callback function that will be invoked for each peer device that was found. An abstract C++ class called `BLEAdvertisedDeviceCallbacks` works for us here. This has a method on it called `onResult()` which will be invoked for each result found during scanning. This method is passed in a reference to an object of type `BLEAdvertisedDevice` that describes the nature of the device that we found.

For example:

```
class MyCallbacks: public BLEAdvertisedDeviceCallbacks {
   void onResult(BLEAdvertisedDevice* pAdvertisedDevice) {
      // Do something with the found device ...
   }
}

BLE::initClient();
```

```
BLESan *pMyScan = BLE::getScan();
pMyScan->setAdvertisedDeviceCallbacks(new MyCallbacks());
pMyScan->start();
```

What this means is that for every device that the scan find, a call to `onResult()` will be made which is passed a `BLEAdvertisedDevice` which describes that device. Now let us look at what a device *may* tell us. The only thing that a device will tell us for sure is its BLE address. Beyond that, everything else is optional. The *possible* attributes that an advertised device may inform us about is large and the following subset have been modeled (so far):

- Appearance
- Manufacturer data
- Name
- RSSI
- Primary service UUID
- Transmit power

Since none of these are mandatory in an advertisement, for any given advertised device we learn about, we can't immediately ask about the value of the property. Instead, we have methods that tell us which properties are present and, if present **then** we can ask for its value.

For example, in our processing logic we may code:

```
if (pAdvertisedDevice->hasServiceUUID) {
   BLEUUID service = pAdvertisedDevice->getServiceUUID();
   if (service.equals(BLEUUID((uint16_t)0x1802) {
      // we found a usefule device …
   }
}
```

(The above is merely an example of the logic that can be employed).

From all of this … we end up with the identification of a device that was of interest to us and since we want to be a BLE Client, this will presumably mean the device to which we wish to connect. The key item in the advertisement now becomes the device's address which we can obtain through a call to `getAddress()`.

Now we can turn our attention to connecting to the device.

A BLE Client is modeled as the `BLEClient` class. We obtain an unconnected instance of this by asking the device for one:

```
BLEClient* pMyClient = BLE::createClient();
```

Next we request a connection to the target device.

```
pMyClient->connect(address);
```

… and that's it. The `connect()` is a blocking call and, on return, we are connected. However, that isn't the end of the story. Just connecting to a peer is usually not enough. Now we want to read and write the values that the remote BLE server is hosting. If we think back to our understanding of the story, the remote server has one or more services and each service has one or more characteristics and it is these characteristics that hold the values. This means that it doesn't make sense to say "I want to read the value of the remote BLE server" … instead we must say "I want to read the value of a named remote characteristic". However, even that is not enough. There may be multiple services on the BLE server each of which have their own characteristic which may be of the same characteristic type. Thus we end up with the final concept of "I want to read the value of a named remote characteristic that is owned by a named service". From this notion, we now get to introduce two more models. Those are the `BLERemoteService` and `BLERemoteCharacteristic`.

Once we have connected to a BLE Server, we can request a reference to the `BLERemoteService` that models a service on that server:

```
BLERemoteService* pMyRemoteService = pClient->getService(serviceUUID);
```

and once we have a reference to the service, we can ask *that* service for a reference to the desired characteristic.

```
BLERemoteCharacteristic* pMyRemoteCharacteristic =
   pMyRemoteService->getCharacteristic(characteristicUUID);
```

and ... finally … we can work with the values.

```
std::string myValue = pMyRemoteCharacteristic->readValue();
```

to read the value and …

```
pMyRemoteCharacteristic->writeValue("abc");
```

to write a new value. If all that seems like a lot of work … lets put it together in context and see:

```
BLEClient* pMyClient = BLE::createClient();
pMyClient->connect(address);
BLERemoteService* pMyRemoteService = pClient->getService(serviceUUID);
BLERemoteCharacteristic* pMyRemoteCharacteristic =
   pMyRemoteService->getCharacteristic(characteristicUUID);
std::string myValue = pMyRemoteCharacteristic->readValue();
```

… and that's it. Hopefully you see that is pretty elegant (and if not, *please* contact me and we'll see if we can't improve on it … but come bearing suggestions for what can be improved).