Bluetooth Low Energy

In this document

Key classes

- `BluetoothGatt`
- `BluetoothGattCallback`
- `BluetoothGattCharacteristic`
- `BluetoothGattService`

Related samples

- Android BluetoothLeGatt Sample
- Android BluetoothAdvertisements Sample

See Also

- Best Practices for Bluetooth Development (video)

VIDEO
DevBytes: Bluetooth Low Energy API

Android 4.3 (API level 18) introduces built-in platform support for Bluetooth Low Energy (BLE) in the *central role* and provides APIs that apps can use to discover devices, query for services, and transmit information.

Common use cases include the following:

- Transferring small amounts of data between nearby devices.
- Interacting with proximity sensors like Google Beacons to give users a customized experience based on their current location.

In contrast to Classic Bluetooth, Bluetooth Low Energy (BLE) is designed to provide significantly lower power consumption. This allows Android apps to communicate with BLE devices that have stricter power requirements, such as proximity sensors, heart rate monitors, and fitness devices.

Key Terms and Concepts

Here is a summary of key BLE terms and concepts:

- **Generic Attribute Profile (GATT)**—The GATT profile is a general specification for sending and receiving short pieces of data known as

"attributes" over a BLE link. All current Low Energy application profiles are based on GATT.

  - The Bluetooth SIG defines many profiles for Low Energy devices. A profile is a specification for how a device works in a particular application. Note that a device can implement more than one profile. For example, a device could contain a heart rate monitor and a battery level detector.

- **Attribute Protocol (ATT)**—GATT is built on top of the Attribute Protocol (ATT). This is also referred to as GATT/ATT. ATT is optimized to run on BLE devices. To this end, it uses as few bytes as possible. Each attribute is uniquely identified by a Universally Unique Identifier (UUID), which is a standardized 128-bit format for a string ID used to uniquely identify information. The *attributes* transported by ATT are formatted as *characteristics* and *services*.

- **Characteristic**—A characteristic contains a single value and 0-n descriptors that describe the characteristic's value. A characteristic can be thought of as a type, analogous to a class.

- **Descriptor**—Descriptors are defined attributes that describe a characteristic value. For example, a descriptor might specify a human-readable description, an acceptable range for a characteristic's value, or a unit of measure that is specific to a characteristic's value.

- **Service**—A service is a collection of characteristics. For example, you could have a service called "Heart Rate Monitor" that includes characteristics such as "heart rate measurement." You can find a list of existing GATT-based profiles and services on bluetooth.org.

Roles and Responsibilities

Here are the roles and responsibilities that apply when an Android device interacts with a BLE device:

- Central vs. peripheral. This applies to the BLE connection itself. The device in the central role scans, looking for advertisement, and the device in the peripheral role makes the advertisement.

- GATT server vs. GATT client. This determines how two devices talk to each other once they've established the connection.

To understand the distinction, imagine that you have an Android phone and an activity tracker that is a BLE device. The phone supports the central role; the activity tracker supports the peripheral role (to establish a BLE connection you need one of each—two things that only support peripheral couldn't talk to each other, nor could two things that only support central).

Once the phone and the activity tracker have established a connection, they start transferring GATT metadata to one another. Depending on the kind of data they transfer, one or the other might act as the server. For example, if the activity tracker wants to report sensor data to the phone, it might make sense for the activity tracker to act as the server. If the activity tracker wants to receive updates from the phone, then it might make sense for the phone to act as the server.

In the example used in this document, the Android app (running on an Android device) is the GATT client. The app gets data from the GATT server, which is a BLE heart rate monitor that supports the Heart Rate Profile. But you could alternatively design your Android app to play the GATT server role. See `BluetoothGattServer` for more information.

BLE Permissions

In order to use Bluetooth features in your application, you must declare the Bluetooth permission `BLUETOOTH`. You need this permission to perform any Bluetooth communication, such as requesting a connection, accepting a connection, and transferring data.

If you want your app to initiate device discovery or manipulate Bluetooth settings, you must also declare the `BLUETOOTH_ADMIN` permission. **Note:** If you use the `BLUETOOTH_ADMIN` permission, then you must also have the `BLUETOOTH` permission.

Declare the Bluetooth permission(s) in your application manifest file. For example:

```
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
```

If you want to declare that your app is available to BLE-capable devices only, include the following in your app's manifest:

```
<uses-feature android:name="android.hardware.bluetooth_le" android:required="true"/>
```

However, if you want to make your app available to devices that don't support BLE, you should still include this element in your app's

manifest, but set `required="false"`. Then at run-time you can determine BLE availability by using `PackageManager.hasSystemFeature()`:

```
// Use this check to determine whether BLE is supported on the device. Then
// you can selectively disable BLE-related features.
if (!getPackageManager().hasSystemFeature(PackageManager.FEATURE_BLUETOOTH_LE)) {
    Toast.makeText(this, R.string.ble_not_supported, Toast.LENGTH_SHORT).show();
    finish();
}
```

> **Note:** LE Beacons are often associated with location. In order to use `BluetoothLeScanner` without a filter, you must request the user's permission by declaring either the `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION` permission in your app's manifest file. Without these permissions, scans won't return any results.

Setting Up BLE

Before your application can communicate over BLE, you need to verify that BLE is supported on the device, and if so, ensure that it is enabled. Note that this check is only necessary if `<uses-feature.../>` is set to false.

If BLE is not supported, then you should gracefully disable any BLE features. If BLE is supported, but disabled, then you can request that the user enable Bluetooth without leaving your application. This setup is accomplished in two steps, using the `BluetoothAdapter`.

1. Get the `BluetoothAdapter`

   The `BluetoothAdapter` is required for any and all Bluetooth activity. The `BluetoothAdapter` represents the device's own Bluetooth adapter (the Bluetooth radio). There's one Bluetooth adapter for the entire system, and your application can interact with it using this object. The snippet below shows how to get the adapter. Note that this approach uses `getSystemService()` to return an instance of `BluetoothManager`, which is then used to get the adapter. Android 4.3 (API Level 18) introduces `BluetoothManager`:

   ```
   private BluetoothAdapter mBluetoothAdapter;
   ...
   // Initializes Bluetooth adapter.
   final BluetoothManager bluetoothManager =
           (BluetoothManager) getSystemService(Context.BLUETOOTH_SERVICE);
   mBluetoothAdapter = bluetoothManager.getAdapter();
   ```

2. Enable Bluetooth

   Next, you need to ensure that Bluetooth is enabled. Call `isEnabled()` to check whether Bluetooth is currently enabled. If this method returns false, then Bluetooth is disabled. The following snippet checks whether Bluetooth is enabled. If it isn't, the snippet displays an error prompting the user to go to Settings to enable Bluetooth:

   ```
   // Ensures Bluetooth is available on the device and it is enabled. If not,
   // displays a dialog requesting user permission to enable Bluetooth.
   if (mBluetoothAdapter == null || !mBluetoothAdapter.isEnabled()) {
       Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
       startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
   }
   ```

   > **Note:** The `REQUEST_ENABLE_BT` constant passed to `startActivityForResult(android.content.Intent, int)` is a locally-defined integer (which must be greater than 0) that the system passes back to you in your `onActivityResult(int, int, android.content.Intent)` implementation as the `requestCode` parameter.

Finding BLE Devices

To find BLE devices, you use the `startLeScan()` method. This method takes a `BluetoothAdapter.LeScanCallback` as a parameter. You must implement this callback, because that is how scan results are returned. Because scanning is battery-intensive, you should observe the following guidelines:

- As soon as you find the desired device, stop scanning.

- Never scan on a loop, and set a time limit on your scan. A device that was previously available may have moved out of range, and continuing to scan drains the battery.

The following snippet shows how to start and stop a scan:

```
/**
 * Activity for scanning and displaying available BLE devices.
 */
public class DeviceScanActivity extends ListActivity {

    private BluetoothAdapter mBluetoothAdapter;
    private boolean mScanning;
    private Handler mHandler;

    // Stops scanning after 10 seconds.
    private static final long SCAN_PERIOD = 10000;
    ...
    private void scanLeDevice(final boolean enable) {
        if (enable) {
            // Stops scanning after a pre-defined scan period.
            mHandler.postDelayed(new Runnable() {
                @Override
                public void run() {
                    mScanning = false;
                    mBluetoothAdapter.stopLeScan(mLeScanCallback);
                }
            }, SCAN_PERIOD);

            mScanning = true;
            mBluetoothAdapter.startLeScan(mLeScanCallback);
        } else {
            mScanning = false;
            mBluetoothAdapter.stopLeScan(mLeScanCallback);
        }
        ...
    }
...
}
```

If you want to scan for only specific types of peripherals, you can instead call `startLeScan(UUID[], BluetoothAdapter.LeScanCallback)`, providing an array of `UUID` objects that specify the GATT services your app supports.

Here is an implementation of the `BluetoothAdapter.LeScanCallback`, which is the interface used to deliver BLE scan results:

```
private LeDeviceListAdapter mLeDeviceListAdapter;
...
// Device scan callback.
private BluetoothAdapter.LeScanCallback mLeScanCallback =
        new BluetoothAdapter.LeScanCallback() {
    @Override
    public void onLeScan(final BluetoothDevice device, int rssi,
            byte[] scanRecord) {
        runOnUiThread(new Runnable() {
           @Override
           public void run() {
               mLeDeviceListAdapter.addDevice(device);
               mLeDeviceListAdapter.notifyDataSetChanged();
           }
       });
   }
};
```

> **Note:** You can only scan for Bluetooth LE devices *or* scan for Classic Bluetooth devices, as described in Bluetooth. You cannot scan for both Bluetooth LE and classic devices at the same time.

Connecting to a GATT Server

The first step in interacting with a BLE device is connecting to it— more specifically, connecting to the GATT server on the device. To connect to a GATT server on a BLE device, you use the `connectGatt()` method. This method takes three parameters: a `Context` object, `autoConnect` (boolean indicating whether to automatically connect to the BLE device as soon as it becomes available), and a reference to a `BluetoothGattCallback`:

```
mBluetoothGatt = device.connectGatt(this, false, mGattCallback);
```

This connects to the GATT server hosted by the BLE device, and returns a `BluetoothGatt` instance, which you can then use to conduct GATT client operations. The caller (the Android app) is the GATT client. The `BluetoothGattCallback` is used to deliver results to the client, such as connection status, as well as any further GATT client operations.

In this example, the BLE app provides an activity (`DeviceControlActivity`) to connect, display data, and display GATT services and characteristics supported by the device. Based on user input, this activity communicates with a `Service` called `BluetoothLeService`, which interacts with the BLE device via the Android BLE API:

```
// A service that interacts with the BLE device via the Android BLE API.
public class BluetoothLeService extends Service {
    private final static String TAG = BluetoothLeService.class.getSimpleName();

    private BluetoothManager mBluetoothManager;
    private BluetoothAdapter mBluetoothAdapter;
    private String mBluetoothDeviceAddress;
    private BluetoothGatt mBluetoothGatt;
    private int mConnectionState = STATE_DISCONNECTED;

    private static final int STATE_DISCONNECTED = 0;
    private static final int STATE_CONNECTING = 1;
    private static final int STATE_CONNECTED = 2;

    public final static String ACTION_GATT_CONNECTED =
            "com.example.bluetooth.le.ACTION_GATT_CONNECTED";
    public final static String ACTION_GATT_DISCONNECTED =
            "com.example.bluetooth.le.ACTION_GATT_DISCONNECTED";
    public final static String ACTION_GATT_SERVICES_DISCOVERED =
            "com.example.bluetooth.le.ACTION_GATT_SERVICES_DISCOVERED";
    public final static String ACTION_DATA_AVAILABLE =
            "com.example.bluetooth.le.ACTION_DATA_AVAILABLE";
    public final static String EXTRA_DATA =
            "com.example.bluetooth.le.EXTRA_DATA";

    public final static UUID UUID_HEART_RATE_MEASUREMENT =
            UUID.fromString(SampleGattAttributes.HEART_RATE_MEASUREMENT);

    // Various callback methods defined by the BLE API.
    private final BluetoothGattCallback mGattCallback =
            new BluetoothGattCallback() {
        @Override
        public void onConnectionStateChange(BluetoothGatt gatt, int status,
                int newState) {
            String intentAction;
            if (newState == BluetoothProfile.STATE_CONNECTED) {
                intentAction = ACTION_GATT_CONNECTED;
                mConnectionState = STATE_CONNECTED;
                broadcastUpdate(intentAction);
                Log.i(TAG, "Connected to GATT server.");
                Log.i(TAG, "Attempting to start service discovery:" +
                        mBluetoothGatt.discoverServices());

            } else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
                intentAction = ACTION_GATT_DISCONNECTED;
                mConnectionState = STATE_DISCONNECTED;
                Log.i(TAG, "Disconnected from GATT server.");
                broadcastUpdate(intentAction);
            }
        }

        @Override
        // New services discovered
```

```
        public void onServicesDiscovered(BluetoothGatt gatt, int status) {
            if (status == BluetoothGatt.GATT_SUCCESS) {
                broadcastUpdate(ACTION_GATT_SERVICES_DISCOVERED);
            } else {
                Log.w(TAG, "onServicesDiscovered received: " + status);
            }
        }

        @Override
        // Result of a characteristic read operation
        public void onCharacteristicRead(BluetoothGatt gatt,
                BluetoothGattCharacteristic characteristic,
                int status) {
            if (status == BluetoothGatt.GATT_SUCCESS) {
                broadcastUpdate(ACTION_DATA_AVAILABLE, characteristic);
            }
        }
     ...
    };
...
}
```

When a particular callback is triggered, it calls the appropriate `broadcastUpdate()` helper method and passes it an action. Note that the data parsing in this section is performed in accordance with the Bluetooth Heart Rate Measurement profile specifications:

```
private void broadcastUpdate(final String action) {
    final Intent intent = new Intent(action);
    sendBroadcast(intent);
}

private void broadcastUpdate(final String action,
                             final BluetoothGattCharacteristic characteristic) {
    final Intent intent = new Intent(action);

    // This is special handling for the Heart Rate Measurement profile. Data
    // parsing is carried out as per profile specifications.
    if (UUID_HEART_RATE_MEASUREMENT.equals(characteristic.getUuid())) {
        int flag = characteristic.getProperties();
        int format = -1;
        if ((flag & 0x01) != 0) {
            format = BluetoothGattCharacteristic.FORMAT_UINT16;
            Log.d(TAG, "Heart rate format UINT16.");
        } else {
            format = BluetoothGattCharacteristic.FORMAT_UINT8;
            Log.d(TAG, "Heart rate format UINT8.");
        }
        final int heartRate = characteristic.getIntValue(format, 1);
        Log.d(TAG, String.format("Received heart rate: %d", heartRate));
        intent.putExtra(EXTRA_DATA, String.valueOf(heartRate));
    } else {
        // For all other profiles, writes the data formatted in HEX.
        final byte[] data = characteristic.getValue();
        if (data != null && data.length > 0) {
            final StringBuilder stringBuilder = new StringBuilder(data.length);
            for(byte byteChar : data)
                stringBuilder.append(String.format("%02X ", byteChar));
            intent.putExtra(EXTRA_DATA, new String(data) + "\n" +
                    stringBuilder.toString());
        }
    }
    sendBroadcast(intent);
}
```

Back in `DeviceControlActivity`, these events are handled by a `BroadcastReceiver`:

```
// Handles various events fired by the Service.
// ACTION_GATT_CONNECTED: connected to a GATT server.
// ACTION_GATT_DISCONNECTED: disconnected from a GATT server.
// ACTION_GATT_SERVICES_DISCOVERED: discovered GATT services.
// ACTION_DATA_AVAILABLE: received data from the device. This can be a
// result of read or notification operations.
private final BroadcastReceiver mGattUpdateReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        final String action = intent.getAction();
        if (BluetoothLeService.ACTION_GATT_CONNECTED.equals(action)) {
            mConnected = true;
            updateConnectionState(R.string.connected);
            invalidateOptionsMenu();
        } else if (BluetoothLeService.ACTION_GATT_DISCONNECTED.equals(action)) {
            mConnected = false;
            updateConnectionState(R.string.disconnected);
            invalidateOptionsMenu();
            clearUI();
        } else if (BluetoothLeService.
                ACTION_GATT_SERVICES_DISCOVERED.equals(action)) {
            // Show all the supported services and characteristics on the
            // user interface.
            displayGattServices(mBluetoothLeService.getSupportedGattServices());
        } else if (BluetoothLeService.ACTION_DATA_AVAILABLE.equals(action)) {
            displayData(intent.getStringExtra(BluetoothLeService.EXTRA_DATA));
        }
    }
};
```

Reading BLE Attributes

Once your Android app has connected to a GATT server and discovered services, it can read and write attributes, where supported. For example, this snippet iterates through the server's services and characteristics and displays them in the UI:

```
public class DeviceControlActivity extends Activity {
    ...
    // Demonstrates how to iterate through the supported GATT
    // Services/Characteristics.
    // In this sample, we populate the data structure that is bound to the
    // ExpandableListView on the UI.
    private void displayGattServices(List<BluetoothGattService> gattServices) {
        if (gattServices == null) return;
        String uuid = null;
        String unknownServiceString = getResources().
                getString(R.string.unknown_service);
        String unknownCharaString = getResources().
                getString(R.string.unknown_characteristic);
        ArrayList<HashMap<String, String>> gattServiceData =
                new ArrayList<HashMap<String, String>>();
        ArrayList<ArrayList<HashMap<String, String>>> gattCharacteristicData
                = new ArrayList<ArrayList<HashMap<String, String>>>();
        mGattCharacteristics =
                new ArrayList<ArrayList<BluetoothGattCharacteristic>>();

        // Loops through available GATT Services.
        for (BluetoothGattService gattService : gattServices) {
            HashMap<String, String> currentServiceData =
                    new HashMap<String, String>();
            uuid = gattService.getUuid().toString();
            currentServiceData.put(
                    LIST_NAME, SampleGattAttributes.
                            lookup(uuid, unknownServiceString));
            currentServiceData.put(LIST_UUID, uuid);
            gattServiceData.add(currentServiceData);

            ArrayList<HashMap<String, String>> gattCharacteristicGroupData =
                    new ArrayList<HashMap<String, String>>();
            List<BluetoothGattCharacteristic> gattCharacteristics =
                    gattService.getCharacteristics();
            ArrayList<BluetoothGattCharacteristic> charas =
                    new ArrayList<BluetoothGattCharacteristic>();
           // Loops through available Characteristics.
            for (BluetoothGattCharacteristic gattCharacteristic :
                    gattCharacteristics) {
                charas.add(gattCharacteristic);
                HashMap<String, String> currentCharaData =
                        new HashMap<String, String>();
                uuid = gattCharacteristic.getUuid().toString();
                currentCharaData.put(
                        LIST_NAME, SampleGattAttributes.lookup(uuid,
                                unknownCharaString));
                currentCharaData.put(LIST_UUID, uuid);
                gattCharacteristicGroupData.add(currentCharaData);
            }
            mGattCharacteristics.add(charas);
            gattCharacteristicData.add(gattCharacteristicGroupData);
         }
    ...
    }
...
}
```

Receiving GATT Notifications

It's common for BLE apps to ask to be notified when a particular characteristic changes on the device. This snippet shows how to set a notification for a characteristic, using the `setCharacteristicNotification()` method:

```
private BluetoothGatt mBluetoothGatt;
BluetoothGattCharacteristic characteristic;
boolean enabled;
...
mBluetoothGatt.setCharacteristicNotification(characteristic, enabled);
...
BluetoothGattDescriptor descriptor = characteristic.getDescriptor(
        UUID.fromString(SampleGattAttributes.CLIENT_CHARACTERISTIC_CONFIG));
descriptor.setValue(BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE);
mBluetoothGatt.writeDescriptor(descriptor);
```

Once notifications are enabled for a characteristic, an `onCharacteristicChanged()` callback is triggered if the characteristic changes on the remote device:

```
@Override
// Characteristic notification
public void onCharacteristicChanged(BluetoothGatt gatt,
        BluetoothGattCharacteristic characteristic) {
    broadcastUpdate(ACTION_DATA_AVAILABLE, characteristic);
}
```

Closing the Client App

Once your app has finished using a BLE device, it should call `close()` so the system can release resources appropriately:

```
public void close() {
    if (mBluetoothGatt == null) {
        return;
    }
    mBluetoothGatt.close();
    mBluetoothGatt = null;
}
```