MEDIATEK

Mediatek SmartDevice Programming Guide (MT2523)

2017.01.04

Outline

- Overview
- How to Use
- SmartDevice SDK

Overview

Overview

- Mediatek SmartDevice Library (wearable SDK) provides interfaces for connecting and communicating with Mediatek wearable device.

  It supports the following connection mode:

  - **SPP**
  - **DOGP** (Data transfer over GATT Profile)

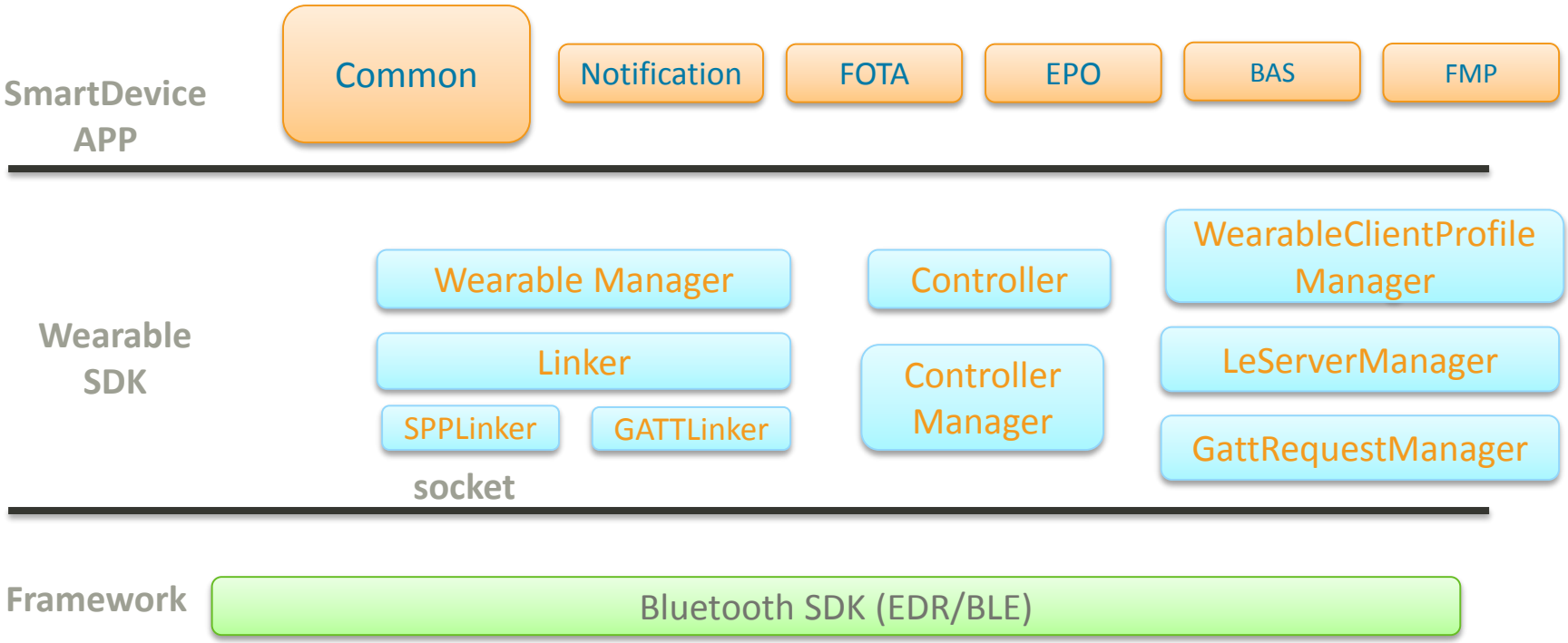
  Mediatek SmartDevice App could send and receive data with wearable device by Bluetooth SPP or DOGP.

This document will introduce how to use wearable SDK for wearable device feature.

Overview

- Mediatek SmartDevice Library Features:
  - Connection & Controller
  - Notifications Push
  - FOTA
  - Add customized BLE servers
  - Add customized BLE clients
  - EPO

Architecture



How to Use

How to Use

- The eclipse is recommended as SmartDevice application development tool.
- Put wearable.jar into libs folder under your Android project, then call library API.
- The Android Bluetooth LE(GATT) API need running Android 4.3(API level 18) or higher.
- SPP API need running Android 4.0(API level 14) or higher.

How to Build

- Because Notification enhance feature must depend on some new Android APIs, the Android 4.4W.2 (API 20) is recommended as Android SDK dependency.
- Import SmartDevice source code in a eclipse Android project, and add wearable.jar to libs folder.
- Build the whole project as Android Application.

SmartDevice APP Source (1/2)

- MTK provides the whole SmartDevice APP source code include wearable.jar.
- User could modify or remove some APP codes, also could add new feature based on wearable SDK.
- The res/xml/wearable_config.xml is wearable SDK parameter customized file.

SmartDevice APP Source (2/2)

- Redesign APP UX

  1. Import SmartDevice APP Source as Android project
  2. Modify source code and resource folder

--- Firmweare upgrade
--- Service/receiver for handling new arrived sms/call/notification
--- MainActivity, MainService, Scan list UI etc.

Customized SDK Parameter

- User could configure wearable SDK parameter by modify wearable_config.xml.
- The wearable_config.xml will be used with WearableManager init method when the SmartDevice process start.
- Sample Code: BTNotificationApplication.onCreate

```
<wearable_config version="1">
    <!--
        SmartDevice APK DOGP send data max size in Write Characteristic setValue method,
        such as 20 ~ 512 byte.
        Aster Watch GATT MTU is 515 byte, APK could increase DOGP transfer rate if APK set larger
        But the feature only is supported by a small part of Android Smart Phone,
        and there will be GATT exception in some Android SPs when set larger gatt_value_size.
    -->
    <int name="gatt_value_size">20</int>

    <!-- Watch BT Noti Src SPP UUID (0x0000FF11 for Larkspur) -->
    <string name="spp_uuid">0000FF01-0000-1000-8000-00805F9B34FB</string>
```

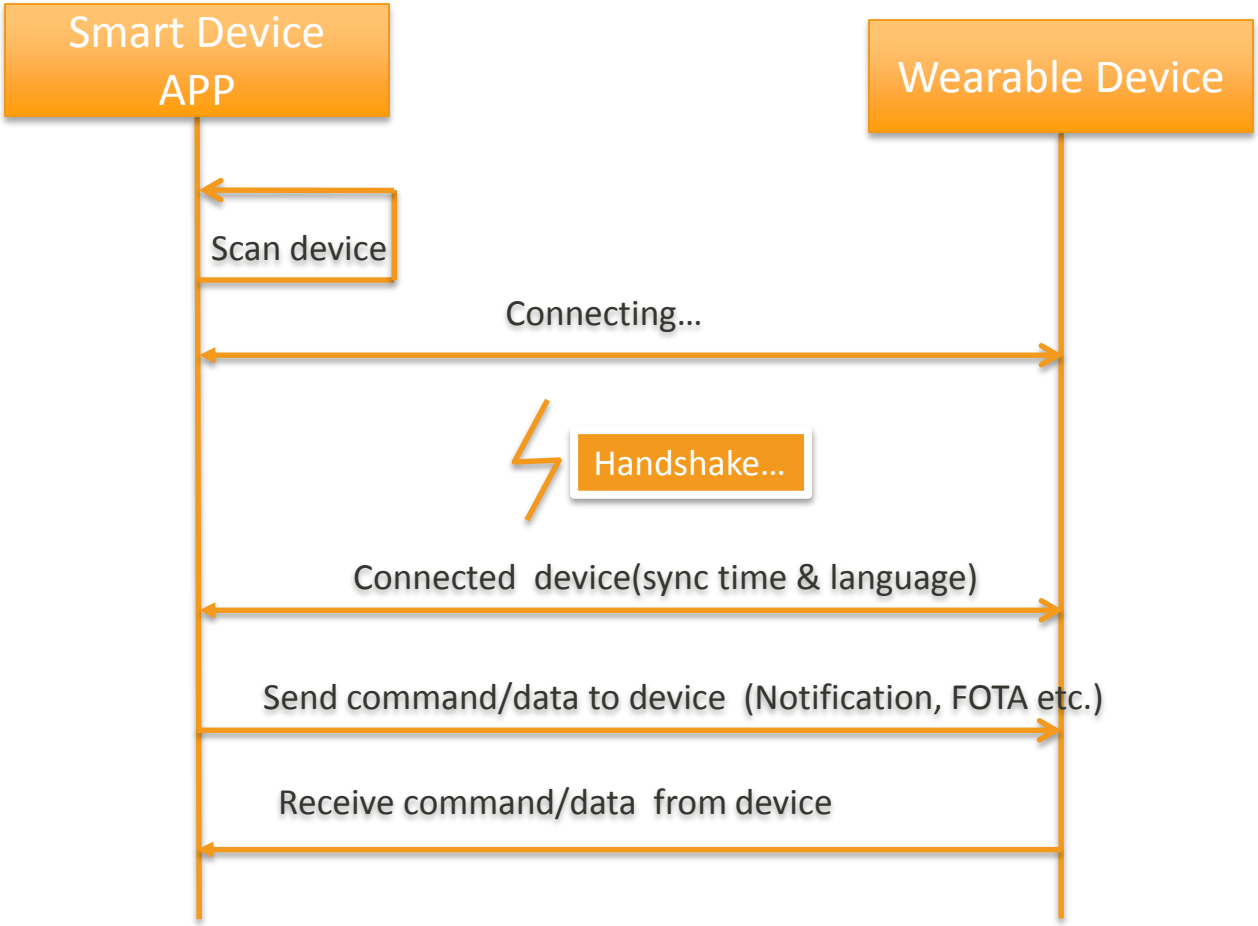
SmartDevice SDK

Connection (1/6)

- The communication between Smart Phone(SP) and wearable, should includes 3 parts: scan, connection, data transfer.
- This below paragraph will introduce how to implement each part by use wearable API.

Connection (2/6)

- The communication flow



Connection (3/6)

Main class and API

- *Controller*
  - Inherit this class and override *send()* & *onReceive()* API to implement send/receive command/data.
- *WearableManager*
  - This can be used to init/scan/connect/disconnect wearable device and notify state/device change .

Connection (4/6)

Init and register WearableListener

- *Init API*
  - init(Boolean isShakeHand, Context appContext, String key, int configResID)
  - Sample Code: BTNotificationApplication onCreate
- *WearableListener*
  - Notify connection/remote device/work mode/discovered device change from WearableManager.
  - API: registerWearableListener unregisterWearableListener

```
public interface WearableListener {

    void onConnectChange(int oldState, int newState);

    void onDeviceChange(BluetoothDevice device);

    void onDeviceScan(BluetoothDevice device);

    void onModeSwitch(int newMode);
}
```

Connection (5/6)

Scan remote device

- *API: WearableManager scan(boolean enable)*
  - Register WearableListener and implement interface method onDeviceScan(BluetoothDevice deviec).
  - Call scan method whether connection mode is SPP or GATT.
- *Sample Code: DeviceScanActivity.java*

Connection (6/6)

Connect & Disconnect

- *API: WearableManager setRemoteDevice, connect, disconnect*
  - WearableManager will callback onDeviceChange(BluetoothDevice) after call setRemoteDevice to select a remote device.
  - Call connect method to establish the connection with wearable device.
  - Disconnect method could disconnect SPP/DOGP connection.

- *API: WearableManager getConnectState isAvailable*
  - getConnectState method could return current connection state.
  - WearableManager will callback onConnectChange to notify connection state change.
  - isAvailable will return true after connect and shake hand successfully.

- *Sample Code: CustomPreference.java*

Controller (1/4)

Data Transfer

- *Controller*
  - Inherit this class and override *send()* & *onReceive()* API to implement send/receive command/data.
- *send(String cmd, byte[] dataBuffer, boolean response, boolean progress, int priority)*
  - cmd: command string sync with wearable device, like “yahooweather yahooweather 1 0 0 ”
  - dataBuffer: data sent to device
  - response: not yet implemented, just set false
  - progress: figure need progress or not, usually set false
  - priority: for real time sending, you can set PRIORITY_HIGH, usually set PRIORITY_NORMAL

Controller (2/4)

Data Transfer

- *onReceive(byte[] dataBuffer)*
  - Received and decode the command or data received.
- *onConnectionStateChange(int state)*
  - Receive connection state change.
- *Controller(String tag, int cmdType)*
  - The controller tag must be unique.
  - The cmdType should be *CMD_9*.

Controller (3/4)

Sample Code

- Sync language Flow:
  1. Wearable request
  2. SP received
  3. SP decode
  4. SP get language
  5. SP send data

```
public class LanguageController extends Controller {
        public static final String LANGUAGE_SENDER = "language";
        public static final String LANGUAGE_RECEIVER = "mtk_language";
……
        public LanguageController () {
            super("LanguageController ", CMD_9);
            HashSet<String> receivers = new HashSet<String>();
            receivers.add(LANGUAGE_RECEIVER);
            super.setReceiverTags(receivers);
          }
…….
        public void sendLanguageCmd() {
            String lan = Locale.getDefault() .toString();
        String cmd = "language mtk_language 1 0 " + lan.length() + " "
        try {
                  super.send(cmd, lan.getBytes(), false, false, PRIORITY_NORMAL);
        } catch (Exception e) {
                 e.printStackTrace();
            }
          }
……
        public void onReceive(byte[] dataBuffer) {// "language mtk_language 0 3 1 0"
        String command = new String(dataBuffer);
            String[] commands = command.split(" ");
          if (commands[1].equals(LANGUAGE_RECEIVER)) {
              if (Integer.valueOf(commands[4]) == 1) {
                 sendLanguageCmd();
              }
            }
        }
}
```

Controller (4/4)

Extensible command format

- Data Format: (Wearable Deivce->SP)

- Data Format: (SP->Wearable Deivce)

cmd: Extensible command type to communicate
sender: APP ID in wearable device
receiver: APP ID in smart phone
err_code: indicate wearable device return code if error happened
data_type: data type, buffer or file
data_len: data length
data: data

Notifications Push

- *NotificationController*
  - Subclass of Controller, use this class to send notification.
  - Send normal type notification API:

    sendNotfications(String appId, CharSequence packageName, CharSequence tickerText, long when, String[] textList)

    sendNotfications(NotificationData notificationData)
  - Send message type notification API:

    sendSmsMessage(String msgbody, String address)
  - Send missed call type notification API:

    sendCallMessage(String phoneNum, String sender, String content, int count)
  - Send low battery type notification API:

    sendLowBatteryMessage(String title, String content, String appId, String value)

FOTA (1/3)

- *FotaOperator*
  - Send command to remote device
  - Send Firmware data to remote device
  - Notify the result from remote device
- *IFotaOperatorCallback*
  - Receive data from remote device
  - Receive status change
- *FotaVersion*
  - The version of the remote device
  - Including SW version, module, brand, dev_id….

FOTA (2/3)

- *FotaOperator*
  - registerFotaCallback
    - Register a IFotaOperatorCallback to monitor the state change and data receiving
  - unregisterFotaCallback
    - Unregister the registered callback, then no any information will be received
  - sendFotaTypeCheckCommand
    - Send type check command to remote device
    - IFotaOperatorCallback#onFotaTypeReceived will be received

FOTA (3/3)

- *FotaOperator*
  - sendFotaVersionGetCommand
    - Send command to get remote device version
    - IFotaOperatorCallback#onFotaVersionReceived will be received
  - sendFotaCustomerInfoGetCommand
    - Send command to get remote device customized information
    - IFotaOperatorCallback#onCustomerInfoReceived will be received
  - sendFotaFirmwareData
    - Send firmware file to remote device according to the file path or file URI
    - While sending the file, IFotaOperatorCallback#onProgress will notify the sending progress

Add customized BLE servers

- Implement one or more LeServers
  - The interface include 3 APIs
    - List<BluetoothGattService> getHardCodeProfileServices()
      - You can prepare your GATT services in this method. The services should be completed, include necessary characteristics and descriptors.
    - BluetoothGattServerCallback getGattServerCallback()
      - LeServer should implement a BluetoothGattServerCallback, when GATT server events coming, the methods in the callback will be called
    - setBluetoothGattServer(BluetoothGattServer server)
      - If your LeServer need use BluetoothGattServer, you may need save this instance.
- Call LeServerManager.addLeServers() method to register your LeServers
- Please reference the file "src/com/mtk/leprofiles/fmpserver/FmpGattServer.java"

Add customized BLE clients

- Extends one or more WearableClientProfile
  - This class extends BluetoothGattCallback, so you can override some callback method
  - If you want to receive the callback about a characteristic or descriptor, you should use addUuids() to mark the UUIDs of the char/desc.
  - If you want to receive the callback of readRssi, you should use enableRssi to mark it.
- After you prepared your WearableClientProfile, call WearableClientProfileManager. registerWearableClientProfile() to register it.
- Suggest calling GattRequestManager.getInstance().readCharacteristic, writeCharacteristic, readDescriptor, writeDescriptor instead of calling these functions in BluetoothGatt
- Please reference the file "src/com/mtk/leprofiles/fmpclient/FmpGattClient.java"