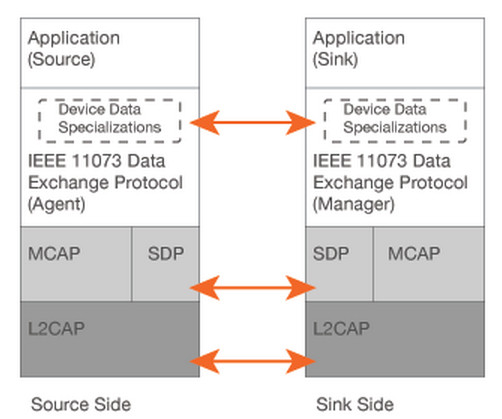
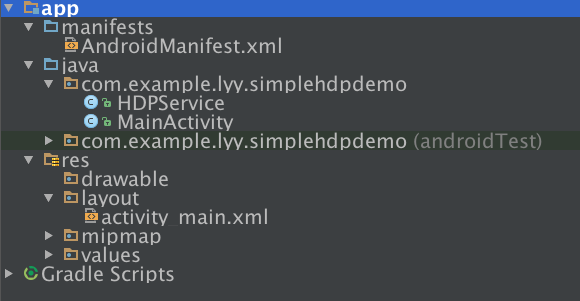
# Android蓝牙健康设备开发：Health Device Profile(HDP)

小项目：用Android手机跟蓝牙血压计通信。在网上查了很多资料，发现有许多文章是讲解Android蓝牙开发，但是在中文社区缺少针对实现HDP profile的蓝牙健康设备的文章，所以整理了相关知识和代码做一个总结。做了一点微小的贡献，谢谢大家。  
1、相关概念  
HDP：Health Device Profile顾名思义是针对**蓝牙健康设备**（如蓝牙血压计、蓝牙体重秤）的一个profile，由蓝牙技术联盟（Bluetooth SIG）发布。并不是所有蓝牙设备都可以采用HDP，只有**经典蓝牙（BR/EDR）设备**才可以。低功耗蓝牙设备，即蓝牙4.0及以上，采用的是GATT等profile。



上图是HDP的协议栈。在通信时，蓝牙健康设备作为source，Android手机作为sink。图中L2CAP、SDP、MCAP是蓝牙通信的**底层协议**，中间是IEEE11073协议。**11073是IEEE发布的一个健康设备的协议簇**，其下包含了许多不同种类的健康设备协议，如11073-10407是血压计的协议。在数据传输时，手机与健康设备根据此协议来发送请求、解析数据等。最上层的application在本文中自然指的就是Android app。  
Android蓝牙模块：android.bluetooth是蓝牙的开发包，里面包含了所有蓝牙相关的类，无论是经典蓝牙还是低功耗蓝牙。其中，本文重点关注的是**BluetoothHealth**这个类，还用到了一些蓝牙基础类BluetoothAdapter BluetoothDevice等，在这里就不展开介绍了。BluetoothHealth是在API14时引入的，所以系统版本高于14的Android手机都可以与采用HDP的健康设备通信。BluetoothHealth中包括了与HDP设备建立通信的API，具体请参见[官方文档](https://developer.android.com/reference/android/bluetooth/BluetoothHealth.html)。在官方文档中有一个建立通信的流程：  
1、调用[getProfileProxy(Context, BluetoothProfile.ServiceListener, int)](https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html#getProfileProxy(android.content.Context,%20android.bluetooth.BluetoothProfile.ServiceListener,%20int))来获取代理对象的连接。  
2、创建BluetoothHealthCallback回调，调用[registerSinkAppConfiguration(String, int, BluetoothHealthCallback)](https://developer.android.com/reference/android/bluetooth/BluetoothHealth.html#registerSinkAppConfiguration(java.lang.String,%20int,%20android.bluetooth.BluetoothHealthCallback))注册一个sink端的应用程序配置。  
3、将手机与健康设备配对，这一步一般在手机的“设置”中完成。  
4、使用[connectChannelToSource(BluetoothDevice, BluetoothHealthAppConfiguration)](https://developer.android.com/reference/android/bluetooth/BluetoothHealth.html#connectChannelToSource(android.bluetooth.BluetoothDevice,%20android.bluetooth.BluetoothHealthAppConfiguration))来建立一个与健康设备的通信channel。有的设备会自动建立通信，不需要在代码中调用这个方法。第二步中的回调会指示channel的状态变化。  
5、用ParcelFileDescriptor来读取健康设备传来的数据，并根据IEEE 11073来解析数据。  
6、通信结束后，关闭通信channel，注销应用程序配置。  
看完这个流程是不是一脸懵逼？不要慌，这里官方文档实在太抽象了，必须要配合实际的代码才能看懂。  
2、demo代码  
项目的目标是与经典蓝牙血压计进行通信，获取血压计测量的数值和日期。这里血压计的型号是A&D UA-767PBT-C。项目中的代码是以github上[这个项目](https://github.com/andengineering/A-D-HDP-Android-Demo)为基础，根据需求进行修改而实现的。



项目结构

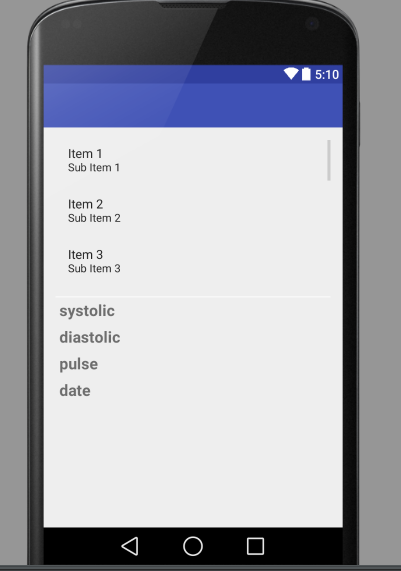
上图是项目的结构，非常简单，只有一个activity和一个service。activity与用户交互，service绑定在activity中与蓝牙血压计建立通信、交换数据。  
首先AndroidManifest.xml中注册蓝牙权限：

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

<uses-permission android:name="android.permission.BLUETOOTH\_ADMIN"/>

## MainActivity

主要作用是显示血压计传过来的数据，以及绑定服务。  
activity的布局如下：



demo.png

上半部分是一个ArrayList,用于显示手机所配对的蓝牙设备的名称和Mac地址。下半部分会显示一次血压测量的参数：收缩压、舒张压、心率、测量时间。

在MainActivity中一些重要私有变量：

private static final int REQUEST\_ENABLE\_BT = 1; //用于打开手机蓝牙

private static final int HEALTH\_PROFILE\_SOURCE\_DATA\_TYPE = 0x1007; //IEEE 11073中规定的血压数据类型

private BluetoothAdapter mBluetoothAdapter;

private Messenger mHealthService; //用于与service通信

private boolean mHealthServiceBound; //用于判断service是否与此activity绑定

打开蓝牙：

if (!mBluetoothAdapter.isEnabled()) {

Intent enableIntent = new Intent(

BluetoothAdapter.ACTION\_REQUEST\_ENABLE);

startActivityForResult(enableIntent, REQUEST\_ENABLE\_BT);

} else {

initialize();

}

启动服务:

// Sets up communication with HDPService.

private ServiceConnection mConnection = new ServiceConnection() {

public void onServiceConnected(ComponentName name, IBinder service) {

mHealthServiceBound = true;

Message msg = Message.obtain(null,

HDPService.MSG\_REG\_CLIENT);

msg.replyTo = mMessenger;

mHealthService = new Messenger(service);

try {

mHealthService.send(msg);

//register blood pressure data type

sendMessage(HDPService.MSG\_REG\_HEALTH\_APP,

HEALTH\_PROFILE\_SOURCE\_DATA\_TYPE);

} catch (RemoteException e) {

Log.w(TAG, "Unable to register client to service.");

e.printStackTrace();

}

}

public void onServiceDisconnected(ComponentName name) {

mHealthService = null;

mHealthServiceBound = false;

}

};

private void initialize() {

// Starts health service.

Intent intent = new Intent(this, HDPService.class);

//startService(intent);

bindService(intent, mConnection, Context.BIND\_AUTO\_CREATE);

}

用handler和message来实现activity与service的通信：

private Handler mIncomingHandler = new Handler() {

@Override

public void handleMessage(Message msg) {

switch (msg.what) {

......

}

}

};

private final Messenger mMessenger = new Messenger(mIncomingHandler);

## HDPService

这个service是核心部分，手机与蓝牙血压计的通信就是在此实现的。  
service中部分重要私有变量:

private BluetoothHealthAppConfiguration mHealthAppConfig; //BluetoothHealthAppConfiguration

private BluetoothAdapter mBluetoothAdapter; //BluetoothAdapter

private BluetoothHealth mBluetoothHealth; //代理对象

在service中按照官方文档的流程一步一步实现与蓝牙血压计的通信。  
1、获取代理对象：

public void onCreate() {

super.onCreate();

mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();

if (mBluetoothAdapter == null || !mBluetoothAdapter.isEnabled()) {

// Bluetooth adapter isn't available. The client of the service is supposed to

// verify that it is available and activate before invoking this service.

stopSelf();

return;

}

if (!mBluetoothAdapter.getProfileProxy(this, mBluetoothServiceListener,

BluetoothProfile.HEALTH)) {

Toast.makeText(this, "HDP not available",

Toast.LENGTH\_LONG).show();

stopSelf();

return;

}

}

代码中的mBluetoothServiceListener是一个回调，如果getProfileProxy方法返回为真，就进入回调。在回调中获取了代理对象属于BluetoothHealth类的proxy。

private final BluetoothProfile.ServiceListener mBluetoothServiceListener =

new BluetoothProfile.ServiceListener() {

public void onServiceConnected(int profile, BluetoothProfile proxy) {

if (profile == BluetoothProfile.HEALTH) {

mBluetoothHealth = (BluetoothHealth) proxy;

Log.d(TAG, "onServiceConnected to profile: " + profile);

}

}

public void onServiceDisconnected(int profile) {

if (profile == BluetoothProfile.HEALTH) {

mBluetoothHealth = null;

}

Log.d(TAG, "onServiceDisconnected");

}

};

2、注册BluetoothHealthAppConfiguration:

mBluetoothHealth.registerSinkAppConfiguration(TAG, dataType, mHealthCallback);

private final BluetoothHealthCallback mHealthCallback = new BluetoothHealthCallback() {

// Callback to handle application registration and unregistration events. The service

// passes the status back to the UI client.

public void onHealthAppConfigurationStatusChange(BluetoothHealthAppConfiguration config,

int status) {

if (status == BluetoothHealth.APP\_CONFIG\_REGISTRATION\_FAILURE) {

mHealthAppConfig = null;

sendMessage(STATUS\_HEALTH\_APP\_REG, RESULT\_FAIL, null);

Log.d(TAG, "APP\_CONFIG\_REGISTRATION\_FAILURE");

} else if (status == BluetoothHealth.APP\_CONFIG\_REGISTRATION\_SUCCESS) {

mHealthAppConfig = config;

sendMessage(STATUS\_HEALTH\_APP\_REG, RESULT\_OK, null);

Log.d(TAG, "APP\_CONFIG\_REGISTRATION\_SUCCESS");

} else if (status == BluetoothHealth.APP\_CONFIG\_UNREGISTRATION\_FAILURE ||

status == BluetoothHealth.APP\_CONFIG\_UNREGISTRATION\_SUCCESS) {

sendMessage(STATUS\_HEALTH\_APP\_UNREG,

status == BluetoothHealth.APP\_CONFIG\_UNREGISTRATION\_SUCCESS ?

RESULT\_OK : RESULT\_FAIL, null);

Log.d(TAG, "UNREGISTRATION");

}

}

// Callback to handle channel connection state changes.

// Note that the logic of the state machine may need to be modified based on the HDP device.

// When the HDP device is connected, the received file descriptor is passed to the

// ReadThread to read the content.

public void onHealthChannelStateChange(BluetoothHealthAppConfiguration config,

BluetoothDevice device, int prevState, int newState, ParcelFileDescriptor fd,

int channelId) {

// if (Log.isLoggable(TAG, Log.DEBUG))

Log.d(TAG, String.format("prevState\t%d ----------> newState\t%d",

prevState, newState));

if (prevState == BluetoothHealth.STATE\_CHANNEL\_CONNECTING &&

newState == BluetoothHealth.STATE\_CHANNEL\_CONNECTED) {

if (config.equals(mHealthAppConfig)) {

mChannelId = channelId;

sendMessage(STATUS\_CREATE\_CHANNEL, RESULT\_OK, null);

(new ReadThread(fd)).start();

} else {

sendMessage(STATUS\_CREATE\_CHANNEL, RESULT\_FAIL, null);

}

} else if (prevState == BluetoothHealth.STATE\_CHANNEL\_CONNECTING &&

newState == BluetoothHealth.STATE\_CHANNEL\_DISCONNECTED) {

sendMessage(STATUS\_CREATE\_CHANNEL, RESULT\_FAIL, null);

} else if (newState == BluetoothHealth.STATE\_CHANNEL\_DISCONNECTED) {

Log.d(TAG, "I'm in State Channel Disconnected.");

if (config.equals(mHealthAppConfig)) {

sendMessage(STATUS\_DESTROY\_CHANNEL, RESULT\_OK, null);

} else {

sendMessage(STATUS\_DESTROY\_CHANNEL, RESULT\_FAIL, null);

}

} else if (prevState == BluetoothHealth.STATE\_CHANNEL\_DISCONNECTED &&

newState == BluetoothHealth.STATE\_CHANNEL\_CONNECTED) {

if (config.equals(mHealthAppConfig)) {

mChannelId = channelId;

sendMessage(STATUS\_CREATE\_CHANNEL, RESULT\_OK, null);

(new ReadThread(fd)).start();

} else {

sendMessage(STATUS\_CREATE\_CHANNEL, RESULT\_FAIL, null);

}

}

}

};

mHealthCallback是其中的关键，有两个方法onHealthAppConfigurationStatusChange和onHealthChannelStateChange。  
onHealthAppConfigurationStatusChange回调会监听AppConfiguration的状态，会在registerSinkAppConfiguration函数返回后调用。  
onHealthChannelStateChange回调会监听与source端（在这里也就是血压计）通信通道的状态变化，如果收到发来的数据，会接收到一个ParcelFileDescriptor用于读取。

3、将手机与健康设备配对  
这一步在手机的系统设置中进行，不必严格按照这个流程的顺序，可以事先配对好。

4、建立通信通道  
这一步不是所有设备都需要，部分设备会自动建立通道与sink端（手机）进行通信。我们所用的这个蓝牙血压计就是如此，会自动建立通道。  
如果不能自动建立通信，可以主动调用如下代码：

mBluetoothHealth.connectChannelToSource(mDevice, mHealthAppConfig);

mDevice是将要建立通信的source端，可以用mBluetoothAdapter获取与手机配对的任一设备。

5、读取、解析数据  
在与蓝牙设备通信时，我们需要知道一点：通信是双向的，设备会向手机发信息，手机可以也需要向设备发信息。在service中建了两个线程，一个用于读数据，一个用于写数据，通过这两个线程实现与设备的数据交换。

首先看读线程

private class ReadThread extends Thread {

private ParcelFileDescriptor mFd;

public ReadThread(ParcelFileDescriptor fd) {

super();

mFd = fd;

Log.i(TAG, "read thread");

}

@Override

public void run() {

...

}

}

读线程的调用是在回调方法onHealthChannelStateChange中的，当判断通信通道打开时，会开启一个读线程，并且将回调中提供的ParcelFileDescriptor传入读线程的构造方法，用于读取数据。  
具体的读取和解析过程我们来看run方法：

FileInputStream fis = new FileInputStream(mFd.getFileDescriptor());

byte data[] = new byte[300]; //假定有300byte的数据

try {

while (fis.read(data) > -1) {

if (data[0] != (byte) 0x00) {

String test = byte2hex(data);

Log.i(TAG, "test: " + test);

if (data[0] == (byte) 0xE2) {

Log.i(TAG, "E2");

//data\_AR

count = 1;

(new WriteThread(mFd)).start();

try {

sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

count = 2;

(new WriteThread(mFd)).start();

} else if (data[0] == (byte) 0xE7) {

Log.i(TAG, "E7");

if (data[18] == (byte) 0x0d && data[19] == (byte) 0x1d) //fixed report

{

count = 3;

//set invoke id so get correct response

invoke = new byte[]{data[6], data[7]};

//write back response

(new WriteThread(mFd)).start();

//parse data!!

int length = data[21];

Log.i(TAG, "length is " + length);

int number\_of\_data\_packets = data[22 + 5]; //03

//packet\_start starts from handle 0 byte

int packet\_start = 30;

final int SYS\_DIA\_MAP\_DATA = 1;

final int PULSE\_DATA = 2;

final int ERROR\_CODE\_DATA = 3;

for (int i = 0; i < number\_of\_data\_packets; i++) {

int obj\_handle = data[packet\_start + 1];

switch (obj\_handle) {

case SYS\_DIA\_MAP\_DATA:

int sys = byteToUnsignedInt(data[packet\_start + 9]);

int dia = byteToUnsignedInt(data[packet\_start + 11]);

int map = byteToUnsignedInt(data[packet\_start + 13]);

//create team string... 9+13~9+20

Log.i(TAG, "sys is " + sys);

sendMessage(RECEIVED\_SYS, sys, null);

Log.i(TAG, "dia is " + dia);

sendMessage(RECEIVED\_DIA, dia, null);

Log.i(TAG, "map is " + map);

break;

case PULSE\_DATA:

//parse

int pulse = byteToUnsignedInt(data[packet\_start + 5]);

Log.i(TAG, "pulse is " + pulse);

sendMessage(RECEIVED\_PUL, pulse, null);

String month = byteToString(data[packet\_start + 8]);

String day = byteToString(data[packet\_start + 9]);

String year = byteToString(data[packet\_start + 6]) + byteToString(data[packet\_start + 7]);

String hour = byteToString(data[packet\_start + 10]);

String minute = byteToString(data[packet\_start + 11]);

String date = year + "." + month + "." + day + " " + hour + ":" + minute;

Log.v(TAG, "the date is " + date);

sendMessage(RECEIVED\_DATE, 0, date);

break;

case ERROR\_CODE\_DATA:

//need more signal

break;

}

packet\_start += 4 + data[packet\_start + 3]; //4 = ignore beginning four bytes

}

} else {

count = 2;

}

} else if (data[0] == (byte) 0xE4) {

count = 4;

(new WriteThread(mFd)).start();

}

//zero out the data

for (int i = 0; i < data.length; i++) {

data[i] = (byte) 0x00;

}

}

sendMessage(STATUS\_READ\_DATA, 0, null);

}

} catch (IOException ioe) {

}

if (mFd != null) {

try {

mFd.close();

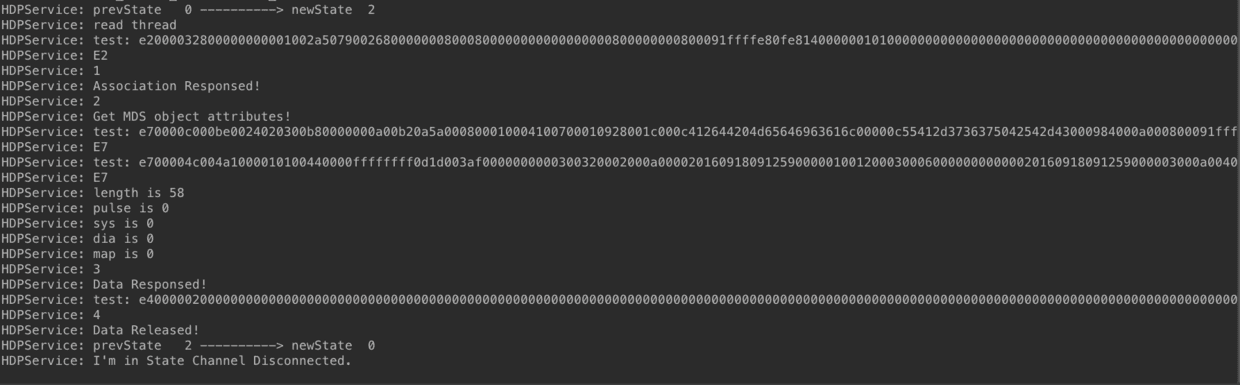
} catch (IOException e) { /\* Do nothing. \*/ }

}

sendMessage(STATUS\_READ\_DATA\_DONE, 0, null);

}

while循环中是对读取数据的解析，这里的解析必须参考IEEE 11073中的内容。这个demo用到了血压计，因此就要使用IEEE 11073－10407（专门针对血压计的协议）。使用其他类型的设备也就要使用相应的IEEE 11073－xxxxx协议。  
回到demo中的数据读取和解析，这是一个根据IEEE 11073－10407进行手机与蓝牙设备互相发送数据的过程，下图是运行的log：



log.png

大致解释一下流程：  
按下测量按钮后，蓝牙血压计发来一个e2开头的数据，这是一个请求连接的信号。  
收到请求连接后，手机用写线程发送给血压计一个回复同意连接，接着再发送一个数据请求血压计的特征。  
血压计发来一个e7开头的数据，回复血压计的特征。  
测量完毕后，血压计发来一个e7开头的数据，是这次测量的报告。  
手机回复确认收到测量数据。  
血压计发来一个e4开头的数据，请求断开连接。  
手机回复确认断开连接。

想要查看详细数据请看IEEE 11073－10407！请看IEEE 11073－10407！请看IEEE 11073－10407！

接下去是写线程：

private class WriteThread extends Thread {

private ParcelFileDescriptor mFd;

public WriteThread(ParcelFileDescriptor fd) {

super();

mFd = fd;

}

@Override

public void run() {

FileOutputStream fos = new FileOutputStream(mFd.getFileDescriptor());

// Association Response [0xE300]

final byte data\_AR[] = new byte[]{(byte) 0xE3, (byte) 0x00,

(byte) 0x00, (byte) 0x2C,

(byte) 0x00, (byte) 0x00,

(byte) 0x50, (byte) 0x79,

(byte) 0x00, (byte) 0x26,

(byte) 0x80, (byte) 0x00, (byte) 0x00, (byte) 0x00,

(byte) 0x80, (byte) 0x00,

(byte) 0x80, (byte) 0x00, (byte) 0x00, (byte) 0x00,

(byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x00,

(byte) 0x80, (byte) 0x00, (byte) 0x00, (byte) 0x00,

(byte) 0x00, (byte) 0x08, //bt add for phone, can be automate in the future

(byte) 0x3C, (byte) 0x5A, (byte) 0x37, (byte) 0xFF,

(byte) 0xFE, (byte) 0x95, (byte) 0xEE, (byte) 0xE3,

(byte) 0x00, (byte) 0x00,

(byte) 0x00, (byte) 0x00,

(byte) 0x00, (byte) 0x00,

(byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x00};

// Presentation APDU [0xE700]

final byte data\_DR[] = new byte[]{(byte) 0xE7, (byte) 0x00,

(byte) 0x00, (byte) 0x12,

(byte) 0x00, (byte) 0x10,

(byte) invoke[0], (byte) invoke[1],

(byte) 0x02, (byte) 0x01,

(byte) 0x00, (byte) 0x0A,

(byte) 0x00, (byte) 0x00,

(byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x00,

(byte) 0x0D, (byte) 0x1D,

(byte) 0x00, (byte) 0x00};

final byte get\_MDS[] = new byte[]{(byte) 0xE7, (byte) 0x00,

(byte) 0x00, (byte) 0x0E,

(byte) 0x00, (byte) 0x0C,

(byte) 0x00, (byte) 0x24,

(byte) 0x01, (byte) 0x03,

(byte) 0x00, (byte) 0x06,

(byte) 0x00, (byte) 0x00,

(byte) 0x00, (byte) 0x00,

(byte) 0x00, (byte) 0x00};

final byte data\_RR[] = new byte[]{(byte) 0xE5, (byte) 0x00,

(byte) 0x00, (byte) 0x02,

(byte) 0x00, (byte) 0x00};

try {

Log.i(TAG, String.valueOf(count));

if (count == 1) {

fos.write(data\_AR);

Log.i(TAG, "Association Responsed!");

} else if (count == 2) {

fos.write(get\_MDS);

Log.i(TAG, "Get MDS object attributes!");

// fos.write(data\_ABORT);

} else if (count == 3) {

fos.write(data\_DR);

Log.i(TAG, "Data Responsed!");

} else if (count == 4) {

fos.write(data\_RR);

Log.i(TAG, "Data Released!");

}

} catch (IOException ioe) {

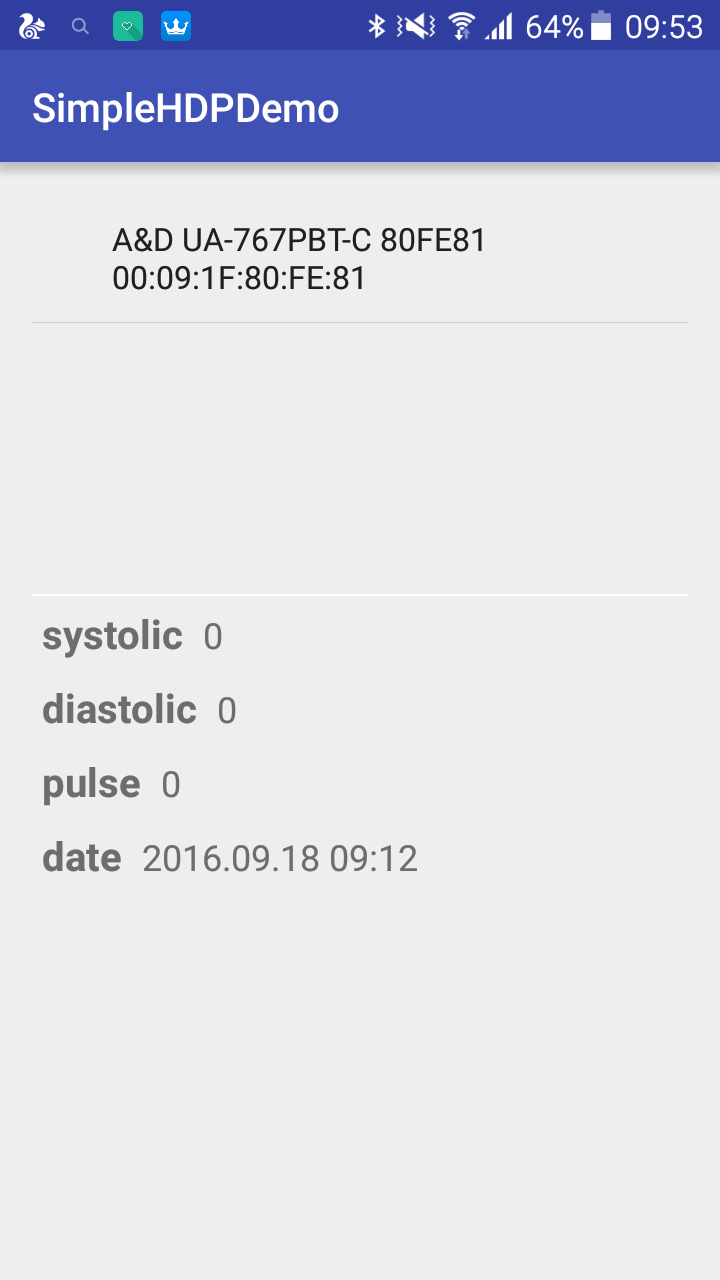
}

}

}

写线程的逻辑非常清晰，就是通过同一个ParcelFileDescriptor向血压计发送数据。发送的数据在这里已经写死，一共四组数据：作用依次是确认连接请求；确认收到测量数据；请求血压计的特征；确认断开连接。

大功告成，收到数据后界面如下：





*/\*\*  
 \* Main user interface for the Sample application. All Bluetooth health-related  
 \* operations happen in {****@link*** *BluetoothHDPService}. This activity passes  
 \* messages to and from the service.  
 \*/*public class BluetoothHDPActivity extends Activity {  
 private static final String *TAG* = "BluetoothHealthActivity";  
  
 // Use the appropriate IEEE 11073 data types based on the devices used.  
 // Below are some examples. Refer to relevant Bluetooth HDP specifications  
 // for detail.  
 // 0x1007 - blood pressure meter  
 // 0x1008 - body thermometer  
 // 0x100F - body weight scale  
 private static final int *HEALTH\_PROFILE\_SOURCE\_DATA\_TYPE* = 0x1007;  
  
 private static final int *REQUEST\_ENABLE\_BT* = 1;  
  
 private TextView mConnectIndicator;  
 private ImageView mDataIndicator;  
 private TextView mStatusMessage;  
  
 private BluetoothAdapter mBluetoothAdapter;  
 private BluetoothDevice[] mAllBondedDevices;  
 private BluetoothDevice mDevice;  
 private int mDeviceIndex = 0;  
 private Resources mRes;  
 private Messenger mHealthService;  
 private boolean mHealthServiceBound;  
  
 // myturnnow  
 private TextView mSys;  
 private TextView mDia;  
 private TextView mPul;  
   
 // version number  
 private TextView mVersion;  
 // Handles events sent by {@link HealthHDPService}.  
 private Handler mIncomingHandler = new Handler() {  
 @Override  
 public void handleMessage(Message msg) {  
 switch (msg.what) {  
 // Application registration complete.  
 case BluetoothHDPService.*STATUS\_HEALTH\_APP\_REG*:  
 mStatusMessage.setText(String.*format*(  
 mRes.getString(R.string.*status\_reg*), msg.arg1));  
 break;  
 // Application unregistration complete.  
 case BluetoothHDPService.*STATUS\_HEALTH\_APP\_UNREG*:  
 mStatusMessage.setText(String.*format*(  
 mRes.getString(R.string.*status\_unreg*), msg.arg1));  
 break;  
 // Reading data from HDP device.  
 case BluetoothHDPService.*STATUS\_READ\_DATA*:  
 mStatusMessage.setText(mRes.getString(R.string.*read\_data*));  
 mDataIndicator.setImageLevel(1);  
 break;  
 // Finish reading data from HDP device.  
 case BluetoothHDPService.*STATUS\_READ\_DATA\_DONE*:  
 mStatusMessage.setText(mRes.getString(R.string.*read\_data\_done*));  
 mDataIndicator.setImageLevel(0);  
 break;  
 // Channel creation complete. Some devices will automatically  
 // establish  
 // connection.  
 case BluetoothHDPService.*STATUS\_CREATE\_CHANNEL*:  
 Log.*d*(*TAG*, "STATUS\_CREATE\_CHANNEl enabled");  
 mStatusMessage.setText(String.*format*(  
 mRes.getString(R.string.*status\_create\_channel*),  
 msg.arg1));  
 mConnectIndicator.setText(R.string.*connected*);  
 break;  
 // Channel destroy complete. This happens when either the device  
 // disconnects or  
 // there is extended inactivity.  
 case BluetoothHDPService.*STATUS\_DESTROY\_CHANNEL*:  
 mStatusMessage.setText(String.*format*(  
 mRes.getString(R.string.*status\_destroy\_channel*),  
 msg.arg1));  
 mConnectIndicator.setText(R.string.*disconnected*);  
 break;  
 case BluetoothHDPService.*RECEIVED\_SYS*:  
 int sys = msg.arg1;  
 Log.*i*(*TAG*, "msg.arg1 @ sys is " + sys);  
 mSys.setText("" + sys);  
 break;  
 case BluetoothHDPService.*RECEIVED\_DIA*:  
 int dia = msg.arg1;  
 mDia.setText("" + dia);  
 Log.*i*(*TAG*, "msg.arg1 @ dia is " + dia);  
 break;  
 case BluetoothHDPService.*RECEIVED\_PUL*:  
 int pul = msg.arg1;  
 Log.*i*(*TAG*, "msg.arg1 @ pulse is " + pul);  
 mPul.setText("" + pul);  
 break;  
 default:  
 super.handleMessage(msg);  
 }  
 }  
 };  
  
 private final Messenger mMessenger = new Messenger(mIncomingHandler);  
  
 @Override  
 public void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
  
 // Check for Bluetooth availability on the Android platform.  
 mBluetoothAdapter = BluetoothAdapter.*getDefaultAdapter*();  
 if (mBluetoothAdapter == null) {  
 Toast.*makeText*(this, R.string.*bluetooth\_not\_available*,  
 Toast.*LENGTH\_LONG*);  
 finish();  
 return;  
 }  
 setContentView(R.layout.*console*);  
 mConnectIndicator = (TextView) findViewById(R.id.*connect\_ind*);  
 mStatusMessage = (TextView) findViewById(R.id.*status\_msg*);  
 mDataIndicator = (ImageView) findViewById(R.id.*data\_ind*);  
 mRes = getResources();  
 mHealthServiceBound = false;  
   
  
 mSys = (TextView) findViewById(R.id.*Systolic*);  
 mDia = (TextView) findViewById(R.id.*Diastolic*);  
 mPul = (TextView) findViewById(R.id.*Pulse*);  
 mVersion = (TextView) findViewById(R.id.*Version*);  
 // mSys.setText("blah");  
 try {  
 String versionName = getPackageManager().getPackageInfo(getPackageName(), 0).versionName;  
 mVersion.setText("V " + versionName);  
 } catch (NameNotFoundException e) {  
 // *TODO Auto-generated catch block* e.printStackTrace();  
 }  
  
 // Initiates application registration through {@link  
 // BluetoothHDPService}.  
 Button registerAppButton = (Button) findViewById(R.id.*button\_register\_app*);  
 registerAppButton.setOnClickListener(new View.OnClickListener() {  
 public void onClick(View v) {  
 sendMessage(BluetoothHDPService.*MSG\_REG\_HEALTH\_APP*,  
 *HEALTH\_PROFILE\_SOURCE\_DATA\_TYPE*);  
 }  
 });  
  
 // Initiates application unregistration through {@link  
 // BluetoothHDPService}.  
 Button unregisterAppButton = (Button) findViewById(R.id.*button\_unregister\_app*);  
 unregisterAppButton.setOnClickListener(new View.OnClickListener() {  
 public void onClick(View v) {  
 sendMessage(BluetoothHDPService.*MSG\_UNREG\_HEALTH\_APP*, 0);  
 }  
 });  
  
 // Initiates channel creation through {@link BluetoothHDPService}. Some  
 // devices will  
 // initiate the channel connection, in which case, it is not necessary  
 // to do this in the  
 // application. When pressed, the user is asked to select from one of  
 // the bonded devices  
 // to connect to.  
 // Button connectButton = (Button)  
 // findViewById(R.id.button\_connect\_channel);  
 // connectButton.setOnClickListener(new View.OnClickListener() {  
 // public void onClick(View v) {  
 // mAllBondedDevices =  
 // (BluetoothDevice[]) mBluetoothAdapter.getBondedDevices().toArray(  
 // new BluetoothDevice[0]);  
 //  
 // if (mAllBondedDevices.length > 0) {  
 // int deviceCount = mAllBondedDevices.length;  
 // if (mDeviceIndex < deviceCount) mDevice =  
 // mAllBondedDevices[mDeviceIndex];  
 // else {  
 // mDeviceIndex = 0;  
 // mDevice = mAllBondedDevices[0];  
 // }  
 // String[] deviceNames = new String[deviceCount];  
 // int i = 0;  
 // for (BluetoothDevice device : mAllBondedDevices) {  
 // deviceNames[i++] = device.getName();  
 // }  
 // SelectDeviceDialogFragment deviceDialog =  
 // SelectDeviceDialogFragment.newInstance(deviceNames, mDeviceIndex);  
 // deviceDialog.show(getFragmentManager(), "deviceDialog");  
 // }  
 // }  
 // });  
  
 // Initiates channel disconnect through {@link BluetoothHDPService}.  
 // Button disconnectButton = (Button)  
 // findViewById(R.id.button\_disconnect\_channel);  
 // disconnectButton.setOnClickListener(new View.OnClickListener() {  
 // public void onClick(View v) {  
 // disconnectChannel();  
 // }  
 // });  
 // registerReceiver(mReceiver, initIntentFilter());  
 }  
  
 // Sets up communication with {@link BluetoothHDPService}.  
 private ServiceConnection mConnection = new ServiceConnection() {  
 public void onServiceConnected(ComponentName name, IBinder service) {  
 mHealthServiceBound = true;  
 Message msg = Message.*obtain*(null,  
 BluetoothHDPService.*MSG\_REG\_CLIENT*);  
 msg.replyTo = mMessenger;  
 mHealthService = new Messenger(service);  
 try {  
 mHealthService.send(msg);  
 } catch (RemoteException e) {  
 Log.*w*(*TAG*, "Unable to register client to service.");  
 e.printStackTrace();  
 }  
 }  
  
 public void onServiceDisconnected(ComponentName name) {  
 mHealthService = null;  
 mHealthServiceBound = false;  
 }  
 };  
  
 @Override  
 protected void onDestroy() {  
 super.onDestroy();  
 if (mHealthServiceBound)  
 unbindService(mConnection);  
 unregisterReceiver(mReceiver);  
 }  
  
 @Override  
 protected void onStart() {  
 super.onStart();  
 // If Bluetooth is not on, request that it be enabled.  
 if (!mBluetoothAdapter.isEnabled()) {  
 Intent enableIntent = new Intent(  
 BluetoothAdapter.*ACTION\_REQUEST\_ENABLE*);  
 startActivityForResult(enableIntent, *REQUEST\_ENABLE\_BT*);  
 } else {  
 initialize();  
 }  
 }  
  
 */\*\*  
 \* Ensures user has turned on Bluetooth on the Android device.  
 \*/* @Override  
 protected void onActivityResult(int requestCode, int resultCode, Intent data) {  
 switch (requestCode) {  
 case *REQUEST\_ENABLE\_BT*:  
 if (resultCode == Activity.*RESULT\_OK*) {  
 initialize();  
 } else {  
 finish();  
 return;  
 }  
 }  
 }  
  
 */\*\*  
 \* Used by {****@link*** *SelectDeviceDialogFragment} to record the bonded Bluetooth  
 \* device selected by the user.  
 \*   
 \** ***@param*** *position  
 \* Position of the bonded Bluetooth device in the array.  
 \*/* public void setDevice(int position) {  
 mDevice = this.mAllBondedDevices[position];  
 mDeviceIndex = position;  
 }  
  
 private void connectChannel() {  
 sendMessageWithDevice(BluetoothHDPService.*MSG\_CONNECT\_CHANNEL*);  
 }  
  
 private void disconnectChannel() {  
 sendMessageWithDevice(BluetoothHDPService.*MSG\_DISCONNECT\_CHANNEL*);  
 }  
  
 private void initialize() {  
 // Starts health service.  
 Intent intent = new Intent(this, BluetoothHDPService.class);  
 startService(intent);  
 bindService(intent, mConnection, Context.*BIND\_AUTO\_CREATE*);  
 }  
  
 // Intent filter and broadcast receive to handle Bluetooth on event.  
 private IntentFilter initIntentFilter() {  
 IntentFilter filter = new IntentFilter();  
 filter.addAction(BluetoothAdapter.*ACTION\_STATE\_CHANGED*);  
 return filter;  
 }  
  
 private final BroadcastReceiver mReceiver = new BroadcastReceiver() {  
 @Override  
 public void onReceive(Context context, Intent intent) {  
 final String action = intent.getAction();  
 if (BluetoothAdapter.*ACTION\_STATE\_CHANGED*.equals(action)) {  
 if (intent.getIntExtra(BluetoothAdapter.*EXTRA\_STATE*,  
 BluetoothAdapter.*ERROR*) == BluetoothAdapter.*STATE\_ON*) {  
 initialize();  
 }  
 }  
 }  
 };  
  
 // Sends a message to {@link BluetoothHDPService}.  
 private void sendMessage(int what, int value) {  
 if (mHealthService == null) {  
 Log.*d*(*TAG*, "Health Service not connected.");  
 return;  
 }  
  
 try {  
 mHealthService.send(Message.*obtain*(null, what, value, 0));  
 } catch (RemoteException e) {  
 Log.*w*(*TAG*, "Unable to reach service.");  
 e.printStackTrace();  
 }  
 }  
  
 // Sends an update message, along with an HDP BluetoothDevice object, to  
 // {@link BluetoothHDPService}. The BluetoothDevice object is needed by the  
 // channel creation  
 // method.  
 private void sendMessageWithDevice(int what) {  
 if (mHealthService == null) {  
 Log.*d*(*TAG*, "Health Service not connected.");  
 return;  
 }  
  
 try {  
 mHealthService.send(Message.*obtain*(null, what, mDevice));  
 } catch (RemoteException e) {  
 Log.*w*(*TAG*, "Unable to reach service.");  
 e.printStackTrace();  
 }  
 }  
  
 */\*\*  
 \* Dialog to display a list of bonded Bluetooth devices for user to select  
 \* from. This is needed only for channel connection initiated from the  
 \* application.  
 \*/* public static class SelectDeviceDialogFragment extends DialogFragment {  
  
 public static SelectDeviceDialogFragment newInstance(String[] names,  
 int position) {  
 SelectDeviceDialogFragment frag = new SelectDeviceDialogFragment();  
 Bundle args = new Bundle();  
 args.putStringArray("names", names);  
 args.putInt("position", position);  
 frag.setArguments(args);  
 return frag;  
 }  
  
 @Override  
 public Dialog onCreateDialog(Bundle savedInstanceState) {  
 String[] deviceNames = getArguments().getStringArray("names");  
 int position = getArguments().getInt("position", -1);  
 if (position == -1)  
 position = 0;  
 return new AlertDialog.Builder(getActivity())  
 .setTitle(R.string.*select\_device*)  
 .setPositiveButton(R.string.*ok*,  
 new DialogInterface.OnClickListener() {  
 public void onClick(DialogInterface dialog,  
 int which) {  
 ((BluetoothHDPActivity) getActivity())  
 .connectChannel();  
 }  
 })  
 .setSingleChoiceItems(deviceNames, position,  
 new DialogInterface.OnClickListener() {  
 public void onClick(DialogInterface dialog,  
 int which) {  
 ((BluetoothHDPActivity) getActivity())  
 .setDevice(which);  
 }  
 }).create();  
 }  
 }  
}

**BluetoothHDPService:**

import java.io.FileInputStream;  
import java.io.FileOutputStream;  
import java.io.IOException;  
  
import android.app.Service;  
import android.bluetooth.BluetoothAdapter;  
import android.bluetooth.BluetoothDevice;  
import android.bluetooth.BluetoothHealth;  
import android.bluetooth.BluetoothHealthAppConfiguration;  
import android.bluetooth.BluetoothHealthCallback;  
import android.bluetooth.BluetoothProfile;  
import android.content.Intent;  
import android.os.Handler;  
import android.os.IBinder;  
import android.os.Message;  
import android.os.Messenger;  
import android.os.ParcelFileDescriptor;  
import android.os.RemoteException;  
import android.util.Log;  
import android.widget.Toast;  
  
*/\*\*  
 \* This Service encapsulates Bluetooth Health API to establish, manage, and disconnect  
 \* communication between the Android device and a Bluetooth HDP-enabled device. Possible HDP  
 \* device type includes blood pressure monitor, glucose meter, thermometer, etc.  
 \*  
 \* As outlined in the  
 \* <a href="http://developer.android.com/reference/android/bluetooth/BluetoothHealth.html">BluetoothHealth</a>  
 \* documentation, the steps involve:  
 \* 1. Get a reference to the BluetoothHealth proxy object.  
 \* 2. Create a BluetoothHealth callback and register an application configuration that acts as a  
 \* Health SINK.  
 \* 3. Establish connection to a health device. Some devices will initiate the connection. It is  
 \* unnecessary to carry out this step for those devices.  
 \* 4. When connected successfully, read / write to the health device using the file descriptor.  
 \* The received data needs to be interpreted using a health manager which implements the  
 \* IEEE 11073-xxxxx specifications.  
 \* 5. When done, close the health channel and unregister the application. The channel will  
 \* also close when there is extended inactivity.  
 \*/*public class BluetoothHDPService extends Service {  
 private static final String *TAG* = "BluetoothHDPService";  
  
 public static final int *RESULT\_OK* = 0;  
 public static final int *RESULT\_FAIL* = -1;  
  
 // Status codes sent back to the UI client.  
 // Application registration complete.  
 public static final int *STATUS\_HEALTH\_APP\_REG* = 100;  
 // Application unregistration complete.  
 public static final int *STATUS\_HEALTH\_APP\_UNREG* = 101;  
 // Channel creation complete.  
 public static final int *STATUS\_CREATE\_CHANNEL* = 102;  
 // Channel destroy complete.  
 public static final int *STATUS\_DESTROY\_CHANNEL* = 103;  
 // Reading data from Bluetooth HDP device.  
 public static final int *STATUS\_READ\_DATA* = 104;  
 // Done with reading data.  
 public static final int *STATUS\_READ\_DATA\_DONE* = 105;  
  
 // Message codes received from the UI client.  
 // Register client with this service.  
 public static final int *MSG\_REG\_CLIENT* = 200;  
 // Unregister client from this service.  
 public static final int *MSG\_UNREG\_CLIENT* = 201;  
 // Register health application.  
 public static final int *MSG\_REG\_HEALTH\_APP* = 300;  
 // Unregister health application.  
 public static final int *MSG\_UNREG\_HEALTH\_APP* = 301;  
 // Connect channel.  
 public static final int *MSG\_CONNECT\_CHANNEL* = 400;  
 // Disconnect channel.  
 public static final int *MSG\_DISCONNECT\_CHANNEL* = 401;  
  
 // Got Reading  
 public static final int *RECEIVED\_SYS* = 500;  
 public static final int *RECEIVED\_DIA* = 501;  
 public static final int *RECEIVED\_PUL* = 502;  
   
 private BluetoothHealthAppConfiguration mHealthAppConfig;  
 private BluetoothAdapter mBluetoothAdapter;  
 private BluetoothHealth mBluetoothHealth;  
 private BluetoothDevice mDevice;  
 private int mChannelId;  
  
 private Messenger mClient;  
 int count = 0;  
 byte invoke[] = new byte[] { (byte) 0x00, (byte) 0x00 };  
   
 // Handles events sent by {@link HealthHDPActivity}.  
 private class IncomingHandler extends Handler {  
 @Override  
 public void handleMessage(Message msg) {  
 switch (msg.what) {  
 // Register UI client to this service so the client can receive messages.  
 case *MSG\_REG\_CLIENT*:  
 Log.*d*(*TAG*, "Activity client registered");  
 mClient = msg.replyTo;  
 break;  
 // Unregister UI client from this service.  
 case *MSG\_UNREG\_CLIENT*:  
 mClient = null;  
 break;  
 // Register health application.  
 case *MSG\_REG\_HEALTH\_APP*:  
 registerApp(msg.arg1);  
 break;  
 // Unregister health application.  
 case *MSG\_UNREG\_HEALTH\_APP*:  
 unregisterApp();  
 break;  
 // Connect channel.  
 case *MSG\_CONNECT\_CHANNEL*:  
 mDevice = (BluetoothDevice) msg.obj;  
 connectChannel();  
 break;  
 // Disconnect channel.  
 case *MSG\_DISCONNECT\_CHANNEL*:  
 mDevice = (BluetoothDevice) msg.obj;  
 disconnectChannel();  
 break;  
 default:  
 super.handleMessage(msg);  
 }  
 }  
 }  
  
 final Messenger mMessenger = new Messenger(new IncomingHandler());  
  
 */\*\*  
 \* Make sure Bluetooth and health profile are available on the Android device. Stop service  
 \* if they are not available.  
 \*/* @Override  
 public void onCreate() {  
 super.onCreate();  
 mBluetoothAdapter = BluetoothAdapter.*getDefaultAdapter*();  
 if (mBluetoothAdapter == null || !mBluetoothAdapter.isEnabled()) {  
 // Bluetooth adapter isn't available. The client of the service is supposed to  
 // verify that it is available and activate before invoking this service.  
 stopSelf();  
 return;  
 }  
 if (!mBluetoothAdapter.getProfileProxy(this, mBluetoothServiceListener,  
 BluetoothProfile.*HEALTH*)) {  
 Toast.*makeText*(this, R.string.*bluetooth\_health\_profile\_not\_available*,  
 Toast.*LENGTH\_LONG*);  
 stopSelf();  
 return;  
 }  
 }  
  
 @Override  
 public int onStartCommand(Intent intent, int flags, int startId) {  
 Log.*d*(*TAG*, "BluetoothHDPService is running.");  
 return *START\_STICKY*;  
 }  
  
 @Override  
 public IBinder onBind(Intent intent) {  
 return mMessenger.getBinder();  
 };  
  
 // Register health application through the Bluetooth Health API.  
 private void registerApp(int dataType) {  
 mBluetoothHealth.registerSinkAppConfiguration(*TAG*, dataType, mHealthCallback);  
 }  
  
 // Unregister health application through the Bluetooth Health API.  
 private void unregisterApp() {  
 mBluetoothHealth.unregisterAppConfiguration(mHealthAppConfig);  
 }  
  
 // Connect channel through the Bluetooth Health API.  
 private void connectChannel() {  
 Log.*i*(*TAG*, "connectChannel()");  
 mBluetoothHealth.connectChannelToSource(mDevice, mHealthAppConfig);  
 }  
  
 // Disconnect channel through the Bluetooth Health API.  
 private void disconnectChannel() {  
 Log.*i*(*TAG*, "disconnectChannel()");  
 mBluetoothHealth.disconnectChannel(mDevice, mHealthAppConfig, mChannelId);  
 }  
  
 // Callbacks to handle connection set up and disconnection clean up.  
 private final BluetoothProfile.ServiceListener mBluetoothServiceListener =  
 new BluetoothProfile.ServiceListener() {  
 public void onServiceConnected(int profile, BluetoothProfile proxy) {  
 if (profile == BluetoothProfile.*HEALTH*) {  
 mBluetoothHealth = (BluetoothHealth) proxy;  
 if (Log.*isLoggable*(*TAG*, Log.*DEBUG*))  
 Log.*d*(*TAG*, "onServiceConnected to profile: " + profile);  
 }  
 }  
  
 public void onServiceDisconnected(int profile) {  
 if (profile == BluetoothProfile.*HEALTH*) {  
 mBluetoothHealth = null;  
 }  
 }  
 };  
  
 private final BluetoothHealthCallback mHealthCallback = new BluetoothHealthCallback() {  
 // Callback to handle application registration and unregistration events. The service  
 // passes the status back to the UI client.  
 public void onHealthAppConfigurationStatusChange(BluetoothHealthAppConfiguration config,  
 int status) {  
 if (status == BluetoothHealth.*APP\_CONFIG\_REGISTRATION\_FAILURE*) {  
 mHealthAppConfig = null;  
 sendMessage(*STATUS\_HEALTH\_APP\_REG*, *RESULT\_FAIL*);  
 } else if (status == BluetoothHealth.*APP\_CONFIG\_REGISTRATION\_SUCCESS*) {  
 mHealthAppConfig = config;  
 sendMessage(*STATUS\_HEALTH\_APP\_REG*, *RESULT\_OK*);  
 } else if (status == BluetoothHealth.*APP\_CONFIG\_UNREGISTRATION\_FAILURE* ||  
 status == BluetoothHealth.*APP\_CONFIG\_UNREGISTRATION\_SUCCESS*) {  
 sendMessage(*STATUS\_HEALTH\_APP\_UNREG*,  
 status == BluetoothHealth.*APP\_CONFIG\_UNREGISTRATION\_SUCCESS* ?  
 *RESULT\_OK* : *RESULT\_FAIL*);  
 }  
 }  
  
 // Callback to handle channel connection state changes.  
 // Note that the logic of the state machine may need to be modified based on the HDP device.  
 // When the HDP device is connected, the received file descriptor is passed to the  
 // ReadThread to read the content.  
 public void onHealthChannelStateChange(BluetoothHealthAppConfiguration config,  
 BluetoothDevice device, int prevState, int newState, ParcelFileDescriptor fd,  
 int channelId) {  
 if (Log.*isLoggable*(*TAG*, Log.*DEBUG*))  
 Log.*d*(*TAG*, String.*format*("prevState\t%d ----------> newState\t%d",  
 prevState, newState));  
 if (prevState == BluetoothHealth.*STATE\_CHANNEL\_DISCONNECTED* &&  
 newState == BluetoothHealth.*STATE\_CHANNEL\_CONNECTED*) {  
 if (config.equals(mHealthAppConfig)) {  
 mChannelId = channelId;  
 sendMessage(*STATUS\_CREATE\_CHANNEL*, *RESULT\_OK*);  
 (new ReadThread(fd)).start();  
 } else {  
 sendMessage(*STATUS\_CREATE\_CHANNEL*, *RESULT\_FAIL*);  
 }  
 } else if (prevState == BluetoothHealth.*STATE\_CHANNEL\_CONNECTING* &&  
 newState == BluetoothHealth.*STATE\_CHANNEL\_DISCONNECTED*) {  
 sendMessage(*STATUS\_CREATE\_CHANNEL*, *RESULT\_FAIL*);  
 } else if (newState == BluetoothHealth.*STATE\_CHANNEL\_DISCONNECTED*) {  
 Log.*d*(*TAG*, "I'm in State Channel Disconnected.");  
 if (config.equals(mHealthAppConfig)) {  
 sendMessage(*STATUS\_DESTROY\_CHANNEL*, *RESULT\_OK*);  
 } else {  
 sendMessage(*STATUS\_DESTROY\_CHANNEL*, *RESULT\_FAIL*);  
 }  
 }  
 }  
 };  
  
 // Sends an update message to registered UI client.  
 private void sendMessage(int what, int value) {  
 if (mClient == null) {  
 Log.*d*(*TAG*, "No clients registered.");  
 return;  
 }  
  
 try {  
 mClient.send(Message.*obtain*(null, what, value, 0));  
 } catch (RemoteException e) {  
 // Unable to reach client.  
 e.printStackTrace();  
 }  
 }  
  
   
 public String byte2hex(byte[] b)  
 {  
 // String Buffer can be used instead  
 String hs = "";  
 String stmp = "";  
  
 for (int n = 0; n < b.length; n++)  
 {  
 stmp = (java.lang.Integer.*toHexString*(b[n] & 0XFF));  
  
 if (stmp.length() == 1)  
 {  
 hs = hs + "0" + stmp;  
 }  
 else  
 {  
 hs = hs + stmp;  
 }  
  
 if (n < b.length - 1)  
 {  
 hs = hs + "";  
 }  
 }  
  
 return hs;  
 }  
   
 public static int byteToUnsignedInt(byte b) {  
 return 0x00 << 24 | b & 0xff;  
 }  
   
 // Thread to read incoming data received from the HDP device. This sample application merely  
 // reads the raw byte from the incoming file descriptor. The data should be interpreted using  
 // a health manager which implements the IEEE 11073-xxxxx specifications.  
 private class ReadThread extends Thread {  
 private ParcelFileDescriptor mFd;  
  
 public ReadThread(ParcelFileDescriptor fd) {  
 super();  
 mFd = fd;  
 }  
   
  
 @Override  
 public void run() {  
 FileInputStream fis = new FileInputStream(mFd.getFileDescriptor());  
 byte data[] = new byte[300];  
 try {  
 while(fis.read(data) > -1) {  
 // At this point, the application can pass the raw data to a parser that  
 // has implemented the IEEE 11073-xxxxx specifications. Instead, this sample  
 // simply indicates that some data has been received.   
 if (data[0] != (byte) 0x00)  
 {  
 String test = byte2hex(data);  
 Log.*i*(*TAG*, test);  
 if(data[0] == (byte) 0xE2){  
 Log.*i*(*TAG*, "E2");  
 //data\_AR  
 count = 1;  
 (new WriteThread(mFd)).start();  
 try {  
 *sleep*(100);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 count = 2;  
 (new WriteThread(mFd)).start();  
 }  
 else if (data[0] == (byte)0xE7){  
 Log.*i*(*TAG*, "E7");  
   
 //work for legacy device...  
 if (data[18] == (byte) 0x0d && data[19] == (byte) 0x1d) //fixed report  
 {  
 count = 3;   
 //set invoke id so get correct response  
 invoke = new byte[] { data[6], data[7] };  
 //write back response  
 (new WriteThread(mFd)).start();   
 //parse data!!  
 int length = data[21];  
 Log.*i*(*TAG*, "length is " + length);  
 // check data-req-id   
// int report\_no = data[22+3];  
 int number\_of\_data\_packets = data[22+5];  
 //packet\_start starts from handle 0 byte  
 int packet\_start = 30;  
 final int SYS\_DIA\_MAP\_DATA = 1;  
 final int PULSE\_DATA = 2;  
 final int ERROR\_CODE\_DATA = 3;  
 for (int i = 0; i < number\_of\_data\_packets; i++)  
 {  
 int obj\_handle = data[packet\_start+1];  
 switch (obj\_handle)  
 {  
 case SYS\_DIA\_MAP\_DATA:  
 int sys = *byteToUnsignedInt*(data[packet\_start+9]);  
 int dia = *byteToUnsignedInt*(data[packet\_start+11]);  
 int map = *byteToUnsignedInt*(data[packet\_start+13]);  
 //create team string... 9+13~9+20   
 Log.*i*(*TAG*, "sys is "+ sys);  
 sendMessage(*RECEIVED\_SYS*, sys);  
 Log.*i*(*TAG*, "dia is "+ dia);  
 sendMessage(*RECEIVED\_DIA*, dia);  
 Log.*i*(*TAG*, "map is "+ map);  
 //test  
// sendMessage(RECEIVED\_MAP, map);  
 break;  
 case PULSE\_DATA:  
 //parse  
 int pulse = *byteToUnsignedInt*(data[packet\_start+5]);  
 Log.*i*(*TAG*, "pulse is " + pulse);  
 sendMessage(*RECEIVED\_PUL*, pulse);  
 break;  
 case ERROR\_CODE\_DATA:  
 //need more signal  
 break;  
 }  
 packet\_start += 4 + data[packet\_start+3]; //4 = ignore beginning four bytes  
 }   
 }  
 else  
 {  
 count = 2;  
 }  
 }  
 else if (data[0] == (byte) 0xE4)  
 {  
 count = 4;  
 (new WriteThread(mFd)).start();  
// sendMessage();  
 }  
 //zero out the data  
 for (int i = 0; i < data.length; i++){  
 data[i] = (byte) 0x00;  
 }  
 }  
 sendMessage(*STATUS\_READ\_DATA*, 0);  
 }  
 } catch(IOException ioe) {}  
 if (mFd != null) {  
 try {  
 mFd.close();  
 } catch (IOException e) { /\* Do nothing. \*/ }  
 }   
 sendMessage(*STATUS\_READ\_DATA\_DONE*, 0);  
 }  
 }  
   
   
 private class WriteThread extends Thread {  
 private ParcelFileDescriptor mFd;  
  
 public WriteThread(ParcelFileDescriptor fd) {  
 super();  
 mFd = fd;  
 }  
  
 @Override  
 public void run() {  
 FileOutputStream fos = new FileOutputStream(mFd.getFileDescriptor());  
// FileInputStream fis = new FileInputStream(mFd.getFileDescriptor());  
// Association Response [0xE300]  
 final byte data\_AR[] = new byte[] { (byte) 0xE3, (byte) 0x00,  
 (byte) 0x00, (byte) 0x2C,   
 (byte) 0x00, (byte) 0x00,  
 (byte) 0x50, (byte) 0x79,  
 (byte) 0x00, (byte) 0x26,  
 (byte) 0x80, (byte) 0x00, (byte) 0x00, (byte) 0x00,  
 (byte) 0x80, (byte) 0x00,  
 (byte) 0x80, (byte) 0x00, (byte) 0x00, (byte) 0x00,  
 (byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x00,  
 (byte) 0x80, (byte) 0x00, (byte) 0x00, (byte) 0x00,  
 (byte) 0x00, (byte) 0x08, //bt add for phone, can be automate in the future  
 (byte) 0x3C, (byte) 0x5A, (byte) 0x37, (byte) 0xFF,   
 (byte) 0xFE, (byte) 0x95, (byte) 0xEE, (byte) 0xE3,  
 (byte) 0x00, (byte) 0x00,  
 (byte) 0x00, (byte) 0x00,  
 (byte) 0x00, (byte) 0x00,   
 (byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x00};  
// Presentation APDU [0xE700]  
 final byte data\_DR[] = new byte[] { (byte) 0xE7, (byte) 0x00,  
 (byte) 0x00, (byte) 0x12,  
 (byte) 0x00, (byte) 0x10,  
 (byte) invoke[0], (byte) invoke[1],  
 (byte) 0x02, (byte) 0x01,  
 (byte) 0x00, (byte) 0x0A,  
 (byte) 0x00, (byte) 0x00,  
 (byte) 0x00, (byte) 0x00, (byte) 0x00, (byte) 0x00,  
 (byte) 0x0D, (byte) 0x1D,  
 (byte) 0x00, (byte) 0x00 };  
   
 final byte get\_MDS[] = new byte[] { (byte) 0xE7, (byte) 0x00,  
 (byte) 0x00, (byte) 0x0E,  
 (byte) 0x00, (byte) 0x0C,  
 (byte) 0x00, (byte) 0x24,  
 (byte) 0x01, (byte) 0x03,  
 (byte) 0x00, (byte) 0x06,  
 (byte) 0x00, (byte) 0x00,  
 (byte) 0x00, (byte) 0x00,  
 (byte) 0x00, (byte) 0x00 };  
   
 final byte data\_RR[] = new byte[] { (byte) 0xE5, (byte) 0x00,  
 (byte) 0x00, (byte) 0x02,  
 (byte) 0x00, (byte) 0x00 };  
   
 final byte data\_RRQ[] = new byte[] { (byte) 0xE4, (byte) 0x00,  
 (byte) 0x00, (byte) 0x02,  
 (byte) 0x00, (byte) 0x00 };  
   
 final byte data\_ABORT[] = new byte[] { (byte) 0xE6, (byte) 0x00,  
 (byte) 0x00, (byte) 0x02,  
 (byte) 0x00, (byte) 0x00 };  
 try {  
 Log.*i*(*TAG*, String.*valueOf*(count));  
 if (count == 1)  
 {  
 fos.write(data\_AR);  
 Log.*i*(*TAG*, "Association Responsed!");  
 }   
 else if (count == 2)  
 {  
 fos.write(get\_MDS);  
 Log.*i*(*TAG*, "Get MDS object attributes!");  
// fos.write(data\_ABORT);  
 }  
 else if (count == 3)   
 {  
 fos.write(data\_DR);  
 Log.*i*(*TAG*, "Data Responsed!");  
 }  
 else if (count == 4)  
 {  
 fos.write(data\_RR);  
 Log.*i*(*TAG*, "Data Released!");  
 }  
 } catch(IOException ioe) {}  
 }  
 }  
   
   
}

<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent" >  
 <LinearLayout android:orientation="vertical"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:layout\_margin="5dp">  
 <TextView android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@string/app\_registration"  
 android:textSize="20sp"  
 android:textStyle="bold" />  
 <LinearLayout android:orientation="horizontal"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:layout\_margin="5dp"  
 android:gravity="center" >  
 <Button android:id="@+id/button\_register\_app"  
 android:layout\_height="wrap\_content"  
 android:layout\_width="wrap\_content"  
 android:minWidth="130dp"  
 android:text="@string/register" />  
 <Button android:id="@+id/button\_unregister\_app"  
 android:layout\_height="wrap\_content"  
 android:layout\_width="wrap\_content"  
 android:minWidth="130dp"  
 android:text="@string/unregister" />  
 </LinearLayout>  
  
 <LinearLayout  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:layout\_margin="5dp"  
 android:orientation="horizontal" >  
 <TextView android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@string/connection\_state"  
 android:textSize="20sp"  
 android:textStyle="bold" />  
 <Space android:layout\_width="10dp"  
 android:layout\_height="0px" />  
 <TextView android:id="@+id/connect\_ind"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@string/disconnected"  
 android:textSize="18sp"/>  
 </LinearLayout>  
  
 <LinearLayout  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:layout\_margin="5dp"  
 android:orientation="horizontal" >  
 <TextView android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@string/data\_ind"  
 android:textSize="20sp"  
 android:textStyle="bold" />  
 <Space android:layout\_width="10dp"  
 android:layout\_height="0px" />  
 <ImageView android:id="@+id/data\_ind"  
 android:layout\_width="20dp"  
 android:layout\_height="20dp"  
 android:layout\_marginTop="4dp"  
 android:contentDescription="@string/desc"  
 android:src="@drawable/led\_indicator" />  
 </LinearLayout>  
  
 <View  
 android:layout\_width="match\_parent"  
 android:layout\_height="1dp"  
 android:background="#FFFFFF" />  
  
 <TextView android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:layout\_margin="5dp"  
 android:text="@string/status\_msg"  
 android:textSize="20sp"  
 android:textStyle="bold" />  
  
 <TextView android:id="@+id/status\_msg"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@string/empty"  
 android:textSize="18sp"  
 android:layout\_margin="10dp"/>  
   
 <View  
 android:layout\_width="match\_parent"  
 android:layout\_height="1dp"  
 android:background="#FFFFFF" />  
 <LinearLayout  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:layout\_margin="5dp"  
 android:orientation="horizontal" >  
 <TextView android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@string/systolic"  
 android:textSize="20sp"  
 android:textStyle="bold" />  
 <Space android:layout\_width="10dp"  
 android:layout\_height="0px" />  
 <TextView android:id="@+id/Systolic"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@string/sys"  
 android:textSize="18sp"/>  
 </LinearLayout>  
  
 <LinearLayout  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:layout\_margin="5dp"  
 android:orientation="horizontal" >  
 <TextView android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@string/diastolic"  
 android:textSize="20sp"  
 android:textStyle="bold" />  
 <Space android:layout\_width="10dp"  
 android:layout\_height="0px" />  
 <TextView android:id="@+id/Diastolic"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@string/dia"  
 android:textSize="18sp"/>  
 </LinearLayout>  
   
 <LinearLayout  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:layout\_margin="5dp"  
 android:orientation="horizontal" >  
 <TextView android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@string/pulse"  
 android:textSize="20sp"  
 android:textStyle="bold" />  
 <Space android:layout\_width="10dp"  
 android:layout\_height="0px" />  
 <TextView android:id="@+id/Pulse"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@string/pul"  
 android:textSize="18sp"/>  
 </LinearLayout>  
 <LinearLayout  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:layout\_margin="5dp"  
 android:gravity="center" >  
 <Space android:layout\_width="10dp"  
 android:layout\_height="20dp" />  
 <TextView android:id="@+id/Version"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@string/version"  
 android:textSize="18sp"/>  
 </LinearLayout>  
 </LinearLayout>  
</ScrollView>

<level-list xmlns:android="http://schemas.android.com/apk/res/android">  
 <item android:maxLevel="0" android:drawable="@drawable/led\_off" />  
 <item android:maxLevel="1" android:drawable="@drawable/led\_on" />  
</level-list>